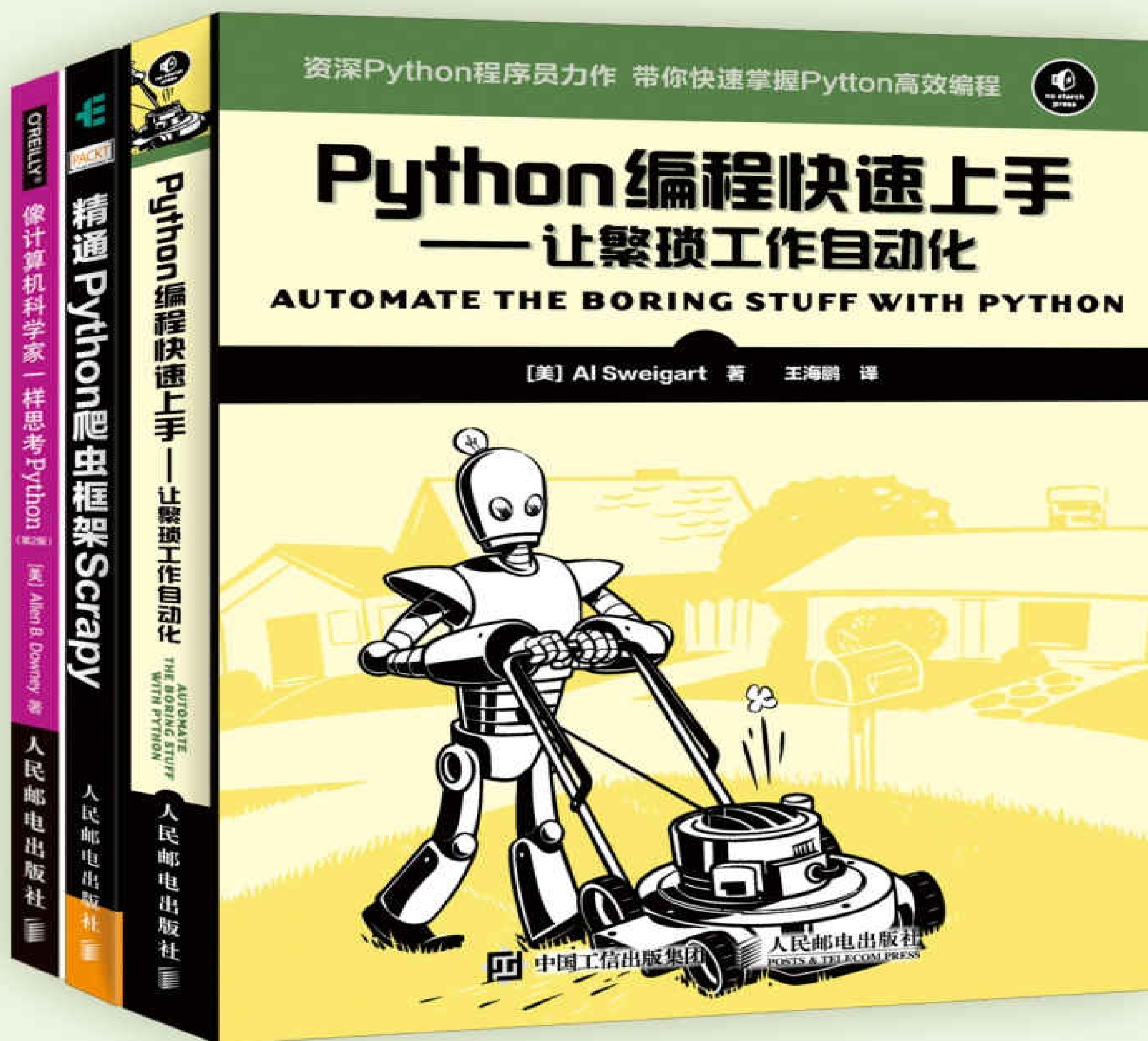
套装共3册

Python编程从认知到实践（第1辑）



人民邮电出版社
POSTS & TELECOM PRESS
异步社区
人民邮电出版社

配套资源下载请访问 异步社区
www.epubit.com

总　目　录

资深Python程序员力作 带你快速掌握Python高效编程

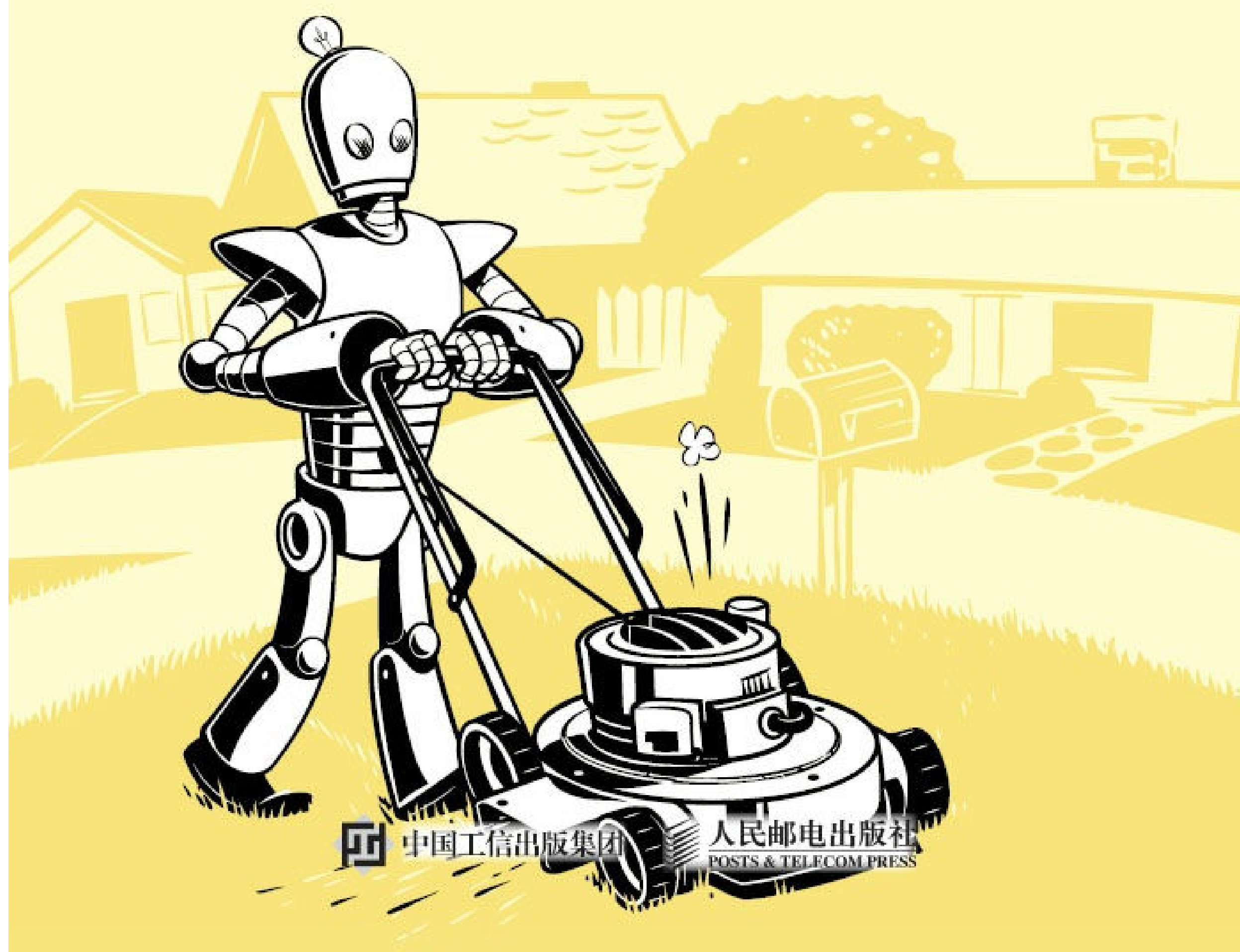


Python编程快速上手

——让繁琐工作自动化

AUTOMATE THE BORING STUFF WITH PYTHON

[美] Al Sweigart 著　　王海鹏 译



目 录

版权信息

书名：Python编程快速上手——让繁琐工作自动化

ISBN：978-7-115-42269-9

- 著　　[美] Al Sweigart

  译　　王海鹏

  责任编辑　陈冀康

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

版权声明

内容提要

如今，人们面临的大多数任务都可以通过编写计算机软件来完成。Python是一种解释型、面向对象、动态数据类型的高级程序设计语言。通过Python编程，我们能够解决现实生活中的很多任务。

本书是一本面向实践的Python编程实用指南。本书的目的，不仅是介绍Python语言的基础知识，而且还通过项目实践教会读者如何应用这些知识和技能。本书的第一部分介绍了基本的Python编程概念，第二部分介绍了一些不同的任务，通过编写Python程序，可以让计算机自动完成它们。第二部分的每一章都有一些项目程序，供读者学习。每章的末尾还提供了一些习题和深入的实践项目，帮助读者巩固所学的知识。附录部分提供了所有习题的解答。

本书适合任何想要通过Python学习编程的读者，尤其适合缺乏编程基础的初学者。通过阅读本书，读者将能利用最强大的编程语言和工具，并且将体会到Python编程的快乐。

作者简介

Al Sweigart是一名软件开发者和技术图书作者，居住在旧金山。Python是他最喜欢的编程语言，他开发了几个开源模块。他的其他著作都在他的网站http://www. inventwithpython.com/ 上。

技术评审者简介

Ari Lacenski是Android应用程序和Python软件开发者。她住在旧金山，她写了一些关于Android编程的文章，放在http://gradlewhy.ghost.io/上，并与Women Who Code合作提供指导。她还是一个民谣吉他手。

致谢

没有很多人的帮助，我不可能写出这样一本书。我想感谢Bill Pollock，我的编辑Laurel Chun、Leslie Shen、Greg Poulos和Jennifer Griffith-Delgado，以及No Starch Press的其他工作人员，感谢他们非常宝贵的帮助。感谢我的技术评审Ari Lacenski，她提供了极好的建议、编辑和支持。

非常感谢Guido van Rossum，以及Python软件基金会的每个人，感谢他们了不起的工作。Python社区是我在业界看到的最佳社区。

最后，我要感谢我的家人和朋友，以及在Shotwell的伙伴，他们不介意我在写这本书时忙碌的生活。干杯！

译者序 会编程的人不一样

这是机器代替人的时代，也是人控制机器的时代。这是程序员的时代，也是非程序员学编程的时代。这是算法的时代，也是编程语言的时代。翻译本书期间，深度学习的人工智能程序AlphaGo以4:1击败了李世石九段。

每一个不会编程的年轻人都应该认真考虑：是不是应该开始学习编程？

学习一门新的语言，总是让人感到畏缩。这让我想起大学时英语老师教的学习方法：听说领先，读写跟上。确实，学语言效果最好的方法就是“用”。本书就遵循了这样的宗旨。本书是面对编程初学者的书，假定读者没有任何编程知识。在简单介绍Python编程语言的基本知识后，就开始用一个接一个的例子，教我们如何用Python来完成一些日常工作，利用计算机这个强大的工具，节省工作时间，提高工作效率，避免手工操作容易带来的错误。

真正的程序员，用编程来解决自己和别人的问题。俄罗斯有一个程序员编写了一个程序，会给老婆发加班短信，会在宿醉不醒时给自己请假，会自动根据邮件恢复客户的数据库，还可以一键远程煮咖啡。加拿大一名零编程基础的农场主，在学习了一门编程课后，开发了一个程序，自动控制拖拉机，配合联合收割机收割谷物。

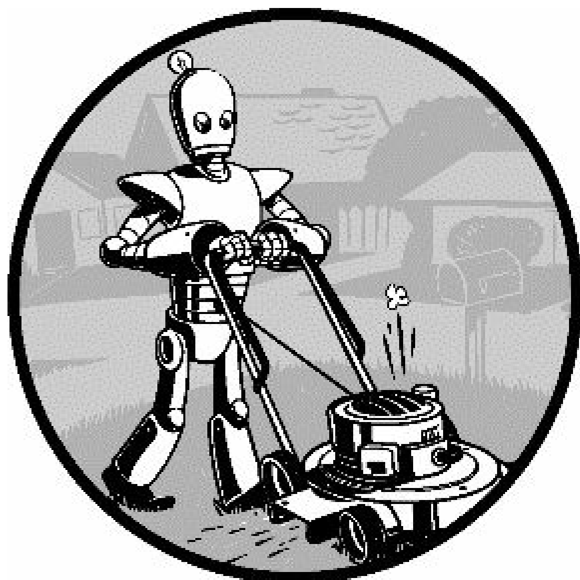
若是已经掌握了其他编程语言，想学习Python，本书也是不错的参考。每一种编程语言，都会提供一种独特的视角，让你对编程有新的认识。我非常喜欢Python没有花括号和分号，程序很“清爽”，符合奥卡姆剃刀原理：如无必要，勿增实体。本书并没有深入介绍面向对象和函数式编程范式，如果想了解Python这方面的内容，请参考其他书籍。

在本书的翻译过程，我自己也在项目中使用Python编程，从中得到许多启发。因此，郑重向大家推荐。翻译中的错误，请不吝指出。

王海鹏

2016年春于上海

前言



“你在2个小时里完成的事，我们3个人要做两天。”21世纪早期，我的大学室友在一个电子产品零售商店工作。商店偶尔会收到一份电子表格，其中包含竞争对手的数千种产品的价格。由3个员工组成的团队，会将这个电子表格打印在一叠厚厚的纸上，然后3个人分一下。针对每个产品价格，他们会查看自己商店的价格，并记录竞争对手价格较低的所有产品。这通常会花几天的时间。

“如果你有打印件的原始文件，我会写一个程序来做这件事。”我的室友告诉他们，当时他看到他们坐在地板上，周围都是散落堆叠的纸张。

几个小时后，他写了一个简短的程序，从文件读取竞争对手的价格，在商店的数据库中找到该产品，并记录竞争对手是否更便宜。他当时还是编程新手，花了许多时间在一本编程书籍中查看文档。实际上程序只花了几秒钟运行。我的室友和他的同事们那天享受了超长的午餐。

这就是计算机编程的威力。计算机就像瑞士军刀，可以用来完成数不清的任务。许多人花上数小时点击鼠标和敲打键盘，执行重复的任务，却没有意识到，如果他们给机器正确的指令，机器就能在几秒钟内完成他们的工作。

本书的读者对象

软件是我们今天使用的许多工具的核心：几乎每个人都使用社交网络来进行交流，许多人的手机中都有连接因特网的计算机，大多数办公室工作都涉及操作计算机来完成工作。因此，对编程人才的需求暴涨。无数的图书、交互式网络教程和开发者新兵训练营，承诺将有雄心壮志的初学者变成软件工程师，获得6位数的薪水。

本书不是针对这些人的。它是针对所有其他的人。

就它本身来说，这本书不会让你变成一个职业软件开发者，就像几节吉他课程不会让你变成一名摇滚巨星。但如果你是办公室职员、管理者、学术研究者，或使用计算机来工作或娱乐的任何人，你将学到编程的基本知识，这样就能将下面这样一些简单的任务自动化：

- 移动并重命名几千个文件，将它们分类，放入文件夹；
- 填写在线表单，不需要打字；
- 在网站更新时，从网站下载文件或复制文本；
- 让计算机向客户发出短信通知；
- 更新或格式化Excel电子表格；
- 检查电子邮件并发出预先写好的回复。

对人来说，这些任务简单，但很花时间。它们通常很琐碎、很特殊，没有现成的软件可以完成。有一点编程知识，就可以让计算机为你完成这些任务。

编码规范

本书没有设计成参考手册，它是初学者指南。编码风格有时候违反最佳实践（例如，有些程序使用全局变量），但这是一种折中，让代码更简单，以便学习。本书的目的是让人们编写用完即抛弃的代码，所以没有太多时间来关注风格和优雅。复杂的编程概念（如面向对象编程、列表推导和生成器），在本书中也没有介绍，因为它们增加了复杂性。编程老手可能会指出，本书中的代码可以修改得更有效率，但本书主要考虑的是用最少的工作量得到能工作的程序。

什么是编程

在电视剧和电影中，常常看到程序员在闪光的屏幕上迅速地输入密码般的一串1和0，但现代编程没有这么神秘。编程只是输入指令让计算机来执行。这些指令可能运算一些数字，修改文本，在文件中查找信息，或通过因特网与其他计算机通信。

所有程序都使用基本指令作为构件块。下面是一些常用的指令，用自然语言的形式来表示：

“做这个，然后做那个。”

“如果这个条件为真，执行这个动作，否则，执行那个动作。”

“按照指定次数执行这个动作。”

“一直做这个，直到条件为真。”

也可以组合这些构件块，实现更复杂的决定。例如，这里有一些编程指令，称为源代码，是用Python编程语言编写的一个简单程序。从头开始，Python软件执行每行代码（有些代码只有在特定条件为真时执行，否则Python会执行另外一些代码），直到到达底部。

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Enter your password.')
typedPassword = input()
❹ if typedPassword == secretPassword:
❺     print('Access granted')
❻     if typedPassword == '12345':
❼         print('That password is one that an idiot puts on their luggage.'
else:
❽     print('Access denied')
```

你可能对编程一无所知，但读了上面的代码，也许就能够合理地猜测它做的事。首先，打开了文件SecretPasswordFile.txt❶，读取了其中的

密码❷。然后，提示用户（通过键盘）输入一个密码❸。比较这两个密码❹，如果它们一样，程序就在屏幕上打印Access granted❺。接下来，程序检查密码是否为12345❻，提示说这可能并不是最好的密码❼。如果密码不一样，程序就在屏幕上打印Access denied❽。

什么是Python

Python指的是Python编程语言（包括语法规则，用于编写被认为是有效的Python代码），以及Python解释器软件，它读取源代码（用python语言编写），并执行其中的指令。Python解释器可以从http://python.org/免费下载，有针对Linux、OS X和Windows的版本。

Python的名字来自于英国超现实主义喜剧团体，而不是来自于蛇。Python程序员被亲切地称为Pythonistas。Monty Python和与蛇相关的引用常常出现在Python的指南和文档中。

程序员不需要知道太多数学

我听到的关于学习编程的最常见的顾虑，就是人们认为这需要很多数学知识。其实，大多数编程需要的数学知识不超过基本算数。实际上，善于编程与善于解决数独问题没有太大差别。

要解决数独问题，数字1到9必须填入9×9的棋盘上每一行、每一列，以及每个3×3的内部方块。通过推导和起始数字的逻辑，你会找到一个答案。例如，在图 1的数独问题中，既然5出现在了左上角，它就不能出现在顶行、最左列，或左上角3×3方块中的其他位置。每次解决一行、一列或一个方块，将为剩下的部分提供更多的数字线索。

仅仅因为数独使用了数字，并不意味着必须精通数学才能求出答案。编程也是这样。就像解决数独问题一样，编程需要将一个问题分解为单个的、详细的步骤。类似地，在调试程序时（即寻找和修复错误），你会耐心地观察程序在做什么，找出缺陷的原因。像所有技能一样，编程越多，你就掌握得越好。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

图1 一个新的数独问题（左边）及其答案（右边）。尽管使用了数字，数独并不需要太多数学知识

编程是创造性活动

编程是一项创造性任务，有点类似于用乐高积木构建一个城堡。你从基本的想法开始，希望城堡看起来像怎样，并盘点可用的积木。然后开始构建。在你完成构建程序后，可以让代码变得更美观，就像对你的城堡那样。

编程与其他创造性活动的不同之处在于，在编程时，你需要的所有原材料都在计算机中，你不需要购买额外的画布、颜料、胶片、纱线、乐高积木或电子器件。在程序写好后，很容易将它在线共享给整个世界。尽管在编程时你会犯错，这项活动仍然很有乐趣。

本书简介

本书的第一部分介绍了基本Python编程概念，第二部分介绍了一些不同的任务，你可以让计算机自动完成它们。第二部分的每一章都有一些项目程序，供你学习。下面简单介绍一下每章的内容。

第一部分：**Python**编程基础

“第 1 章：Python基础”介绍了表达式、Python指令的最基本类型，以及如何使用Python交互式环境来尝试运行代码。

“第 2 章：控制流”解释了如何让程序决定执行哪些指令，以便代码能够智能地响应不同的情况。

“第3章：函数”介绍了如何定义自己的函数，以便将代码组织成可管理的部分。

“第4章：列表”介绍了列表数据类型，解释了如何组织数据。

“第 5 章：字典和结构化数据”介绍了字典数据类型，展示了更强大的数据组织方法。

“第6章：字符串操作”介绍了处理文本数据（在Python中称为字符串）。

第二部分：自动化任务

“第7章：模式匹配与正则表达式”介绍了Python如何用正则表达式

处理字符串，以及查找文本模式。

“第 8 章：读写文件”解释了程序如何读取文本文件的内容，并将信息保存到硬盘的文件中。

“第 9 章：组织文件”展示了Python如何用比手工操作快得多的速度，复制、移动、重命名和删除大量的文件，也解释了压缩和解压缩文件。

“第10章：调试”展示了如何使用Python的缺陷查找和缺陷修复工具。

“第 11 章：从Web抓取信息”展示了如何编程来自动下载网页，解析它们，获取信息。这称为从Web抓取信息。

“第 12 章：处理Excel电子表格”介绍了编程处理Excel电子表格，这样你就不必去阅读它们。如果你必须分析成百上千的文档，这是很有帮助的。

“第13章：处理PDF和Word文档”介绍了编程读取Word和PDF文档。

“第14章：处理CSV文件和JSON数据”解释了如何编程操作CSV和JSON文件。

“第15章：保持时间、计划任务和启动程序”解释了Python程序如何处理时间和日期，如何安排计算机在特定时间执行任务。这一章也展示了Python程序如何启动非Python程序。

“第16章：发送电子邮件和短信”解释了如何编程来发送电子邮件和短信。

“第17章：操作图像”解释了如何编程来操作JPG或PNG这样的图像。

“第18章：用GUI自动化控制键盘和鼠标”解释了如何编程控制鼠标和键盘，自动化鼠标点击和击键。

下载和安装**Python**

可以从http://python.org/downloads/免费下载针对Windows、OS X和Ubuntu的Python版本。如果你从该网站的下载页面下载了最新的版本，本书中的所有程序应该都能工作。

> 注意
>
> 请确保下载Python 3的版本（诸如3.4.0）。本书中的程序将运行在Python 3上，有一部分程序在Python 2上也许不能正常运行。

你需要在下载页面上找到针对64位或32位计算机以及特定操作系统的Python安装程序，所以先要弄清楚你需要哪个安装程序。如果你的计算机是2007年或以后购买的，很有可能是64位的系统。否则，可能是32位的系统，但下面是确认的方法：

- 在Windows上。选择StartControlPanelSystem。检查系统类型是64位或32位。
- 在OS X上，进入Apple菜单，选择About This MacMoreInfoSystemReport Hardware，然后查看Processor Name字段。如果是Intel Core Solo或Intel Core Duo，机器是32位的。如果是其他（包括Intel Core 2 Duo），机器是64位的。
- 在Ubuntu Linux上，打开终端窗口，运行命令uname -m。结果是i686表示是32位，x86_64表示是64位。

在Windows上，下载Python安装程序（文件扩展名是.msi），并双击它。按照安装程序显示在屏幕上的指令来安装Python，步骤如下。

1．选择Install for All Users，然后点击Next。

2．通过点击Next安装到C:\Python34文件夹。

3．再次点击Next，跳过定制Python的部分。

在OS X上，下载适合你的OS X版本的.dmg文件，并双击它。按照安装程序显示在屏幕上的指令来安装Python，步骤如下。

1．当DMG包在一个新窗口中打开时，双击Python.mpkg文件。你可能必须输入管理员口令。

2．点击Continue，跳过欢迎部分，并点击Agree，接受许可证。

3．选择HD Macintosh（或者你的硬盘的名字），并点击Install。

如果使用的是Ubuntu，可以从终端窗口安装Python，步骤如下。

1．打开终端窗口。

2．输入sudo apt-get install python3。

3．输入sudo apt-get install idle3。

4．输入sudo apt-get install python3-pip。

启动IDLE

Python解释器是运行Python程序的软件，而交互式开发环境（IDLE）是输入程序的地方，就像一个字处理软件。现在让我们启动IDLE。

- 在Windows7或更新的版本上，点击屏幕左下角的开始图标，在搜索框中输入IDLE，并选择IDLE（Python GUI）。
- Windows XP上，点击开始按钮，然后选择ProgramsPython 3.4IDLE（Python GUI）。
- 在OS X上，打开Finder窗口，点击Applications，点击Python 3.4，然后点击IDLE的图标。
- 在Ubuntu上，选择ApplicationsAccessoriesTerminal，然后输入idle3（也许你也可以点击屏幕顶部的Applications，选择Programming，然后点击IDLE 3）。

交互式环境

无论你使用什么操作系统，初次出现的IDLE窗口应该基本上是空的，除了类似下面这样的文本：

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64
bit (AMD64)] on win32Type "copyright", "credits" or "license()" for more
information.
>>>
```

这个窗口称为交互式环境。这是让你向计算机输入指令的程序，很像OS X上的终端窗口，或Windows上的命令行提示符。Python的交互式环境让你输入指令，供Python解释器软件来执行。计算机读入你输入的指令，并立即执行它们。

例如，在交互式环境的>>>提示符后输入以下指令：

```
>>> print('Hello world!')
```

在输入该行并按下回车键后，交互式环境将显示以下内容作为响应：

```
>>> print('Hello world!')
Hello world!
```

如何寻求帮助

独自解决编程问题可能比你想的要容易。如果你不相信，就让我们

故意产生一个错误：在交互式环境中输入'42' + 3。现在你不需要知道这条指令是什么意思，但结果看起来应该像这样：

```
>>> '42' + 3
❶ Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
>>>
```

这里出现了错误信息❷，因为Python不理解你的指令。错误信息的traceback部分❶显示了Python遇到困难的特定指令和行号。如果你不知道怎样处理特定的错误信息，就在线查找那条错误信息。在你喜欢的搜索引擎上输入“TypeError: Can't convert 'int' object to str implicitly”（包括引号），你就会看到许多的链接，解释这条错误信息的含义，以及什么原因导致这条错误，如图2所示。

你常常会发现，别人也遇到了同样的问题，而其他乐于助人的人已经回答了这个问题。没有人知道编程的所有方面，所以所有软件开发者的工作，都是每天在寻找技术问题的答案。

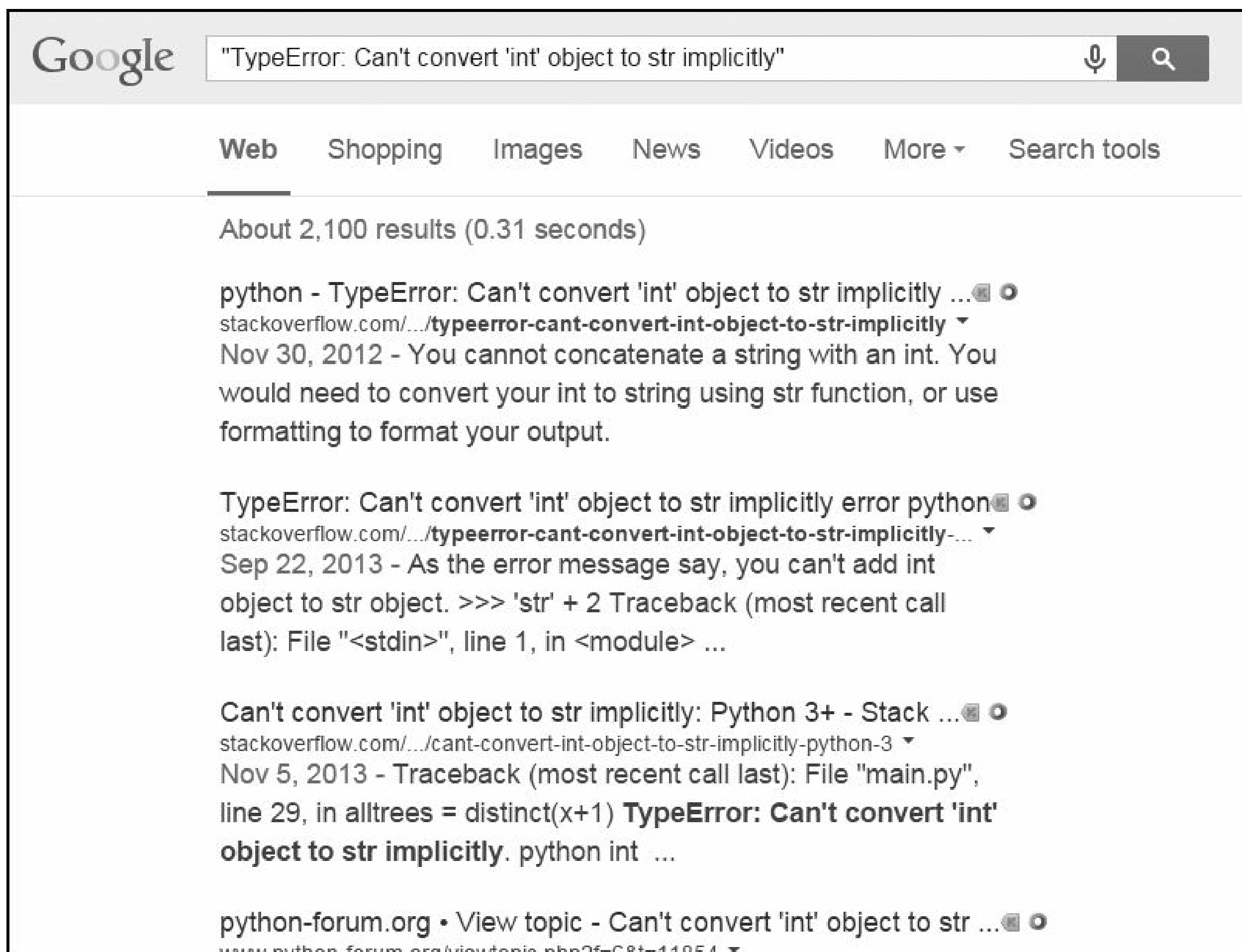


图2 错误信息的Google搜索结果可能非常有用

聪明地提出编程问题

如果不能在线查找到答案，请尝试在Stack Overlow（http://stackoverflow.com/ ）或“learnprogramming”subreddit（http://reddit.com/r/learnprogramming/ ）这样的论坛上提问。但要记住，用聪明的方式提出编程问题，这有助于别人来帮助你。确保阅读这些网站的FAQ（常见问题），了解正确的提问方式。

在提出编程问题时，要记住以下几点。

- 说明你打算做什么，而不只是你做了什么。这让帮助你的人知道你是否走错了路。

- 明确指出发生错误的地方。它是在程序每次启动时发生，还是在你做了某些动作之后？
- 将完整的错误信息和你的代码复制粘贴到http://pastebin.com/或http://gist. github.com/ 。
- 这些网站让你很容易在网上与他人共享大量的代码，而不会丢失任何文本格式。然后你可以将贴出的代码的URL放在电子邮件或论坛帖子中。例如，这里是我贴出的一些代码片段：http://pastebin.com/SzP2DbFx/ 和https://gist.github. com/ asweigart/6912168/ 。
- 解释你为了解决这个问题已经尝试了哪些方法。这会告诉别人你已经做了一些工作来弄清楚状况。
- 列出你使用的Python版本（Python 2解释器和Python3解释器之间有一些重要的区别）。而且，要说明你使用的操作系统和版本。
- 如果错误在你更改了代码之后出现，准确说明你改了什么。
- 说明你是否在每次运行该程序时都能重现该错误，或者它只是在特定的操作执行之后才出现。如果是这样，解释是哪些操作。

也要遵守良好的在线礼节。例如，不要全用大写提问，或者对试图帮助你的人提出无理的要求。

小结

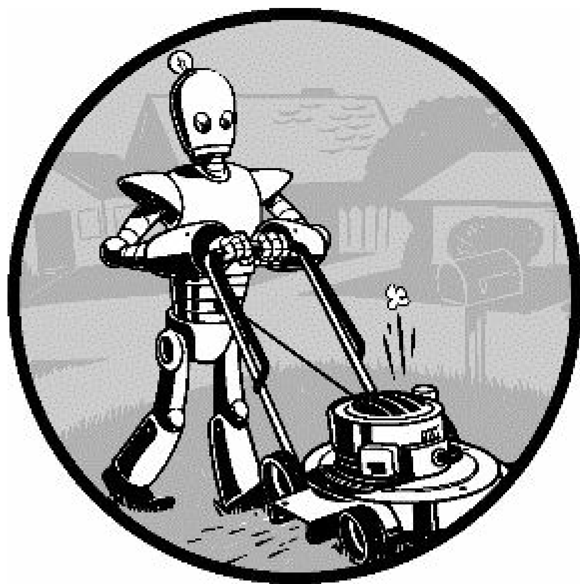
对于大多数人，他们的计算机只是设备，而不是工具。但通过学习如何编程，你就能利用现代社会中最强大的工具，并且你会一直感到快乐。编程不是脑外科手术，业余人士是完全可以尝试或犯错的。

我喜欢帮助人们探索Python。我在自己的博客上编写编程指南（http:// inventwithpython.com/blog/ ），你可以发邮件向我提问（al@inventwithpython.com）。

本书将从零编程知识开始，但你的问题可能超出本书的范围。记住如何有效地提问，知道如何寻找答案，这对你的编程之旅是无价的工具。

让我们开始吧！

第一部分　**Python**编程基础



第1章 Python基础

Python编程语言有许多语法结构、标准库函数和交互式开发环境功能。好在，你可以忽略大多数内容。你只需要学习部分内容，就能编写一些方便的小程序。

但在动手之前，你必须学习一些基本编程概念。就像魔法师培训，你可能认为这些概念既深奥又啰嗦，但有了一些知识和实践，你就能像魔法师一样指挥你的计算机，完成难以置信的事情。

本章有几个例子，我们鼓励你在交互式环境中输入它们。交互式环境让你每次执行一条Python指令，并立即显示结果。使用交互式环境对于了解基本Python指令的行为是很好的，所以你在阅读时要试一下。做过的事比仅仅读过的内容，更令人印象深刻。

1.1 在交互式环境中输入表达式

启动IDLE就运行了交互式环境，这是和Python一起安装的。在Windows上，打开“开始”菜单，选择“All ProgramsPython 3.3”，然后选择“IDLE（Python GUI）”。在OS X上，选择“ApplicationsMacPython 3.3IDLE”。在Ubuntu上，打开新的终端窗口并输入idle3。

一个窗口会出现，包含>>>提示符，这就是交互式环境。在提示符后输入2 + 2，让Python做一些简单的算术。

```
>>> 2 + 2



4
```

IDLE窗口现在应该显示下面这样的文本：

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bi
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>>
```

在Python中，2 + 2称为“表达式”，它是语言中最基本的编程结构。表达式包含“值”（例如2）和“操作符”（例如+），并且总是可以求值（也就是归约）为单个值。这意味着在Python代码中，所有使用表达式的地方，也可以使用一个值。

在前面的例子中，2 + 2被求值为单个值4。没有操作符的单个值也被认为是一个表达式，尽管它求值的结果就是它自己，像下面这样：

```
>>> 2
2
```

错误没关系！

如果程序包含计算机不能理解的代码，就会崩溃，这将导致Python显示错误信息。错误信息并不会破坏你的计算机，所以不要害怕犯错误。“崩溃”只是意味着程序意外地停止执行。

如果你希望对一条错误信息了解更多，可以在网上查找这条信息的准确文本，找到关于这个错误的更多内容。也可以查看http://nostarch.com/automatestuff/，这里有常见的Python错误信息和含义的列表。

Python表达式中也可以使用大量其他操作符。例如，表 1-1 列出了Python的所有数学操作符。

表**1-1** 数学操作符，优先级从高到低

操作符	操作	例子	求值为
**	指数	2 ** 3	8
%	取模/取余数	22 % 8	6
//	整除/商数取整	22 // 8	2
/	除法	22 / 8	2.75
*	乘法	3 * 5	15
-	减法	5 - 2	3
+	加法	2 + 2	4

数学操作符的操作顺序（也称为“优先级”）与数学中类似。**操作符首先求值，接下来是*、/、//和%操作符，从左到右。+和-操作符最后求值，也是从左到右。如果需要，可以用括号来改变通常的优先级。在交互式环境中输入下列表达式：

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
```

```
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 /
 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
```

```
16.0
```

在每个例子中，作为程序员，你必须输入表达式，但Python完成较难的工作，将它求值为单个值。Python将继续求值表达式的各个部分，直到它成为单个值，如图1-1所示。

```
(5 - 1) * ((7 + 1) / (3 - 1))
        ↓
      4 * ((7 + 1) / (3 - 1))
        ↓
      4 * (   8   ) / (3 - 1))
        ↓
      4 * (   8   ) / (   2   )
        ↓
      4 * 4.0
        ↓
       16.0
```

图1-1 表达式求值将它归约为单个值

将操作符和值放在一起构成表达式的这些规则，是 Python 编程语言的基本部分，就像帮助我们沟通的语法规则一样。下面是例子：

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

第二行很难解释，因为它不符合英语的规则。类似地，如果你输入错误的 Python指令，Python也不能理解，就会显示出错误信息，像下面

这样：

```
>>> 5 +




  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2




  File "<stdin>", line 1
    42 + 5 + * 2
              ^
SyntaxError: invalid syntax
```

你总是可以在交互式环境中输入一条指令，检查它是否能工作。不要担心会弄坏计算机：最坏的情况就是Python显示出错信息。专业的软件开发者在编写代码时，常常会遇到错误信息。

1.2 整型、浮点型和字符串数据类型

记住，表达式是值和操作符的组合，它们可以通过求值成为单个值。“数据类型”是一类值，每个值都只属于一种数据类型。表1-2列出了Python中最常见的数据类型。例如，值-2和30属于“整型”值。整型（或int）数据类型表明值是整数。带有小数点的数，如3.14，称为“浮点型”（或float）。请注意，尽管42是一个整型，但42.0是一个浮点型。

表**1-2** 常见数据类型

数据类型	例子

整型	-2, -1, 0, 1, 2, 3, 4, 5
浮点型	-1.25, -1.0, - -0.5, 0.0, 0.5, 1.0, 1.25
字符串	'a', 'aa', 'aaa', 'Hello!', '11 cats'

Python程序也可以有文本值，称为“字符串”，或strs（发音为“stirs”）。总是用单引号（'）包围住字符串（例如'Hello'或'Goodbye cruel world!'），这样Python就知道字符串的开始和结束。甚至可以有没有字符的字符串，称为“空字符串”。第4章更详细地解释了字符串。

如果你看到错误信息SyntaxError: EOL while scanning string literal，可能是忘记了字符串末尾的单引号，如下面的例子所示：

```
>>> 'Hello world!

SyntaxError: EOL while scanning string literal
```

1.3 字符串连接和复制

根据操作符之后的值的数据类型，操作符的含义可能会改变。例如，在操作两个整型或浮点型值时，+是相加操作符。但是，在用于两个字符串时，它将字符串连接起来，成为“字符串连接”操作符。在交互式环境中输入以下内容：

```
>>> 'Alice' + 'Bob'
```

```
'AliceBob'
```

该表达式求值为一个新字符串，包含了两个字符串的文本。但是，如果你对一个字符串和一个整型值使用加操作符，Python就不知道如何处理，它将显示一条错误信息。

```
>>> 'Alice' + 42
Traceback (most recent call last):
   File "<pyshell#26>", line 1, in <module>
     'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

错误信息Can't convert 'int' object to str implicitly表示Python认为，你试图将一个整数连接到字符串'Alice'。代码必须显式地将整数转换为字符串，因为Python不能自动完成转换。（1.6节“程序剖析”在讨论函数时，将解释数据类型转换。）

在用于两个整型或浮点型值时，*操作符表示乘法。但*操作符用于一个字符串值和一个整型值时，它变成了“字符串复制”操作符。在交互式环境中输入一个字符串乘一个数字，看看效果。

```
>>> 'Alice' * 5
```

```
'AliceAliceAliceAliceAlice'
```

该表达式求值为一个字符串，它将原来的字符串重复若干次，次数就是整型的值。字符串复制是一个有用的技巧，但不像字符串连接那样常用。

*操作符只能用于两个数字（作为乘法），或一个字符串和一个整型（作为字符串复制操作符）。否则，Python将显示错误信息。

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Python不理解这些表达式是有道理的：你不能把两个单词相乘，也很难将一个任意字符串复制小数次。

1.4 在变量中保存值

“变量”就像计算机内存中的一个盒子，其中可以存放一个值。如果你的程序稍后将用到一个已求值的表达式的结果，就可以将它保存在一个变量中。

1.4.1 赋值语句

用“赋值语句”将值保存在变量中。赋值语句包含一个变量名、一个等号（称为赋值操作符），以及要存储的值。如果输入赋值语句spam = 42，那么名为spam的变量将保存一个整型值42。

可以将变量看成一个带标签的盒子，值放在其中，如图1-2所示。

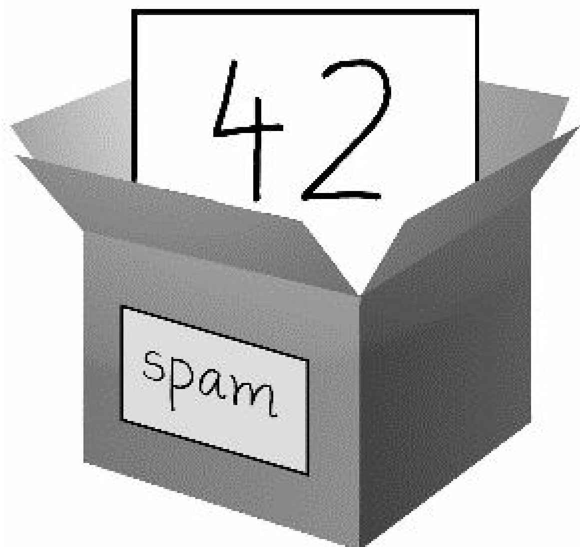


图1-2 spam = 42 就像是告诉程序“变量spam现在有整数42放在里面”

例如，在交互式环境中输入以下内容：

```
❶ >>> spam = 40




  >>> spam
```

```
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

第一次存入一个值，变量就被“初始化”（或创建）❶。此后，可以在表达式中使用它，以及其他变量和值❷。如果变量被赋了一个新值，老值就被忘记了❸。这就是为什么在例子结束时，spam求值为42，而不是40。这称为“覆写”该变量。在交互式环境中输入以下代码，尝试覆写一个字符串：

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

就像图1-3中的盒子，这个例子中的spam变量保存了'Hello'，直到你用'Goodbye'替代它。

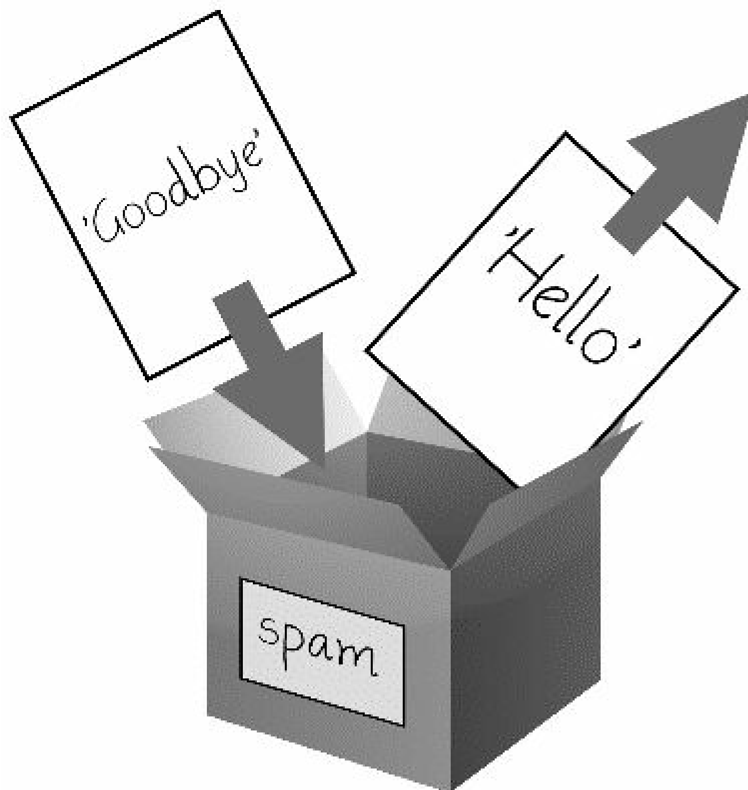


图1-3 如果一个新值赋给变量，老值就被遗忘了

1.4.2 变量名

表1-3中有一些合法变量名的例子。你可以给变量取任何名字，只要它遵守以下3条规则：

1．只能是一个词。

2．只能包含字母、数字和下划线。

3．不能以数字开头。

表**1-3** 有效和无效的变量名

有效的变量名	无效的变量名
balance	current-balance（不允许中划线）

currentBalance	current balanc（不允许空格）
current_balance	4account（不允许数字开头）
_spam	42（不允许数字开头）
SPAM	total_$um（不允许$这样的特殊字符）
account4	'hello'（不允许'这样的特殊字符）

变量名是区分大小写的。这意味着，spam、 SPAM、Spam和sPaM是4个不同的变量。变量用小写字母开头是Python的惯例。

本书的变量名使用了驼峰形式，没有用下划线。也就是说，变量名用lookLikeThis，而不是looking_like_this。一些有经验的程序员可能会指出，官方的Python代码风格PEP 8，即应该使用下划线。我喜欢驼峰式，这没有错，并认为PEP 8本身“愚蠢的一致性是头脑狭隘人士的心魔”：

> “一致地满足风格指南是重要的。但最重要的是，知道何时要不一致，因为有时候风格指南就是不适用。如果有怀疑，请相信自己的最佳判断。”

好的变量名描述了它包含的数据。设想你搬到一间新屋子，搬家纸箱上标的都是“东西”。你永远找不到任何东西！本书的例子和许多Python的文档，使用spam、eggs和bacon等变量名作为一般名称（受到Monty Python的“Spam”短剧的影响），但在你的程序中，具有描述性的名字有助于提高代码可读性。

1.5 第一个程序

虽然交互式环境对于一次运行一条 Python指令很好，但要编写完整的Python程序，就需要在文件编辑器中输入指令。“文件编辑器”类似于Notepad或TextMate这样的文本编辑器，它有一些针对输入源代码的特殊功能。要在IDLE中打开文件编辑器，请选择FileNew ▸Window。

出现的窗口中应该包含一个光标，等待你输入，但它与交互式环境不同。在交互式环境中，按下回车，就会执行Python指令。文件编辑器允许输入许多指令，保存为文件，并运行该程序。下面是区别这两者的方法：

- 交互式环境窗口总是有>>>提示符。
- 文件编辑器窗口没有>>>提示符。

现在是创建第一个程序的时候了！在文件编辑器窗口打开后，输入以下内容：

```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?') # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
❻ print('What is your age?') # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

在输入完源代码后保存它，这样就不必在每次启动IDLE时重新输入。从文件编辑器窗口顶部的菜单，选择File▸Save As。在“Save As”窗口中，在输入框输入hello.py，然后点击“Save”。

在输入程序时，应该过一段时间就保存你的程序。这样，如果计算机崩溃，或者不小心退出了IDLE，也不会丢失代码。作为快捷键，可以在Windows和Linux上按Ctrl-S，在OS X上按⌘-S，来保存文件。

在保存文件后，让我们来运行程序。选择Run▸Run Module，或按下F5键。程序将在交互式环境窗口中运行，该窗口是首次启动IDLE时出现的。记住，必须在文件编辑器窗口中按F5，而不是在交互式环境窗口中。在程序要求输入时，输入你的名字。在交互式环境中，程序输出应该看起来像这样：

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bi
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al







It is good to meet you, Al
The length of your name is:
2
What is your age?
4







You will be 5 in a year.
>>>
```

如果没有更多代码行要执行，Python程序就会“中止”。也就是说，它停止运行。（也可以说Python程序“退出”了。）

可以通过点击窗口上部的X，关闭文件编辑器。要重新加载一个保存了的程序，就在菜单中选择File▸Open。现在请这样做，在出现的窗口中选择hello.py，并点击“Open”按钮。前面保存的程序hello.py应该在文件编辑器窗口中打开。

1.6 程序剖析

新程序在文件编辑器中打开后，让我们快速看一看它用到的Python指令，逐一查看每行代码。

1.6.1 注释

下面这行称为“注释”。

```
❶ # This program says hello and asks for my name.
```

Python会忽略注释，你可以用它们来写程序注解，或提醒自己代码试图完成的事。这一行中，#标志之后的所有文本都是注释。

有时候，程序员在测试代码时，会在一行代码前面加上#，临时删除它。这称为“注释掉”代码。在你想搞清楚为什么程序不工作时，这样做可能有用。稍后，如果你准备还原这一行代码，可以去掉#。

Python也会忽略注释之后的空行。在程序中，想加入空行时就可以加入。这会让你的代码更容易阅读，就像书中的段落一样。

1.6.2 print()函数

print()函数将括号内的字符串显示在屏幕上。

```
❷ print('Hello world!')
  print('What is your name?') # ask for their name
```

代码行print('Hello world!')表示“打印出字符串'Hello world!'的文本”。Python执行到这行时，你告诉Python调用print()函数，并将字符串“传递”给函数。传递给函数的值称为“参数”。请注意，引号没有打印在屏幕上。它们只是表示字符串的起止，不是字符串的一部分。

也可以用这个函数在屏幕上打印出空行，只要调用print()就可以了，括号内没有任何东西。

在写函数名时，末尾的左右括号表明它是一个函数的名字。这就是为什么在本书中你会看到print()，而不是print。第2章更详细地探讨了函数。

1.6.3 input()函数

函数等待用户在键盘上输入一些文本，并按下回车键。

```
❸ myName = input()
```

这个函数求值为一个字符串，即用户输入的文本。前面的代码行将这个字符串赋给变量myName。

你可以认为input()函数调用是一个表达式，它求值为用户输入的任何字符串。如果用户输入'Al'，那么该表达式就求值为myName = 'Al'。

1.6.4 打印用户的名字

接下来的print()调用，在括号间包含表达式'It is good to meet you, ' + myName。

```
❹ print('It is good to meet you, ' + myName)
```

要记住，表达式总是可以求值为一个值。如果'Al'是上一行代码保存在myName中的值，那么这个表达式就求值为'It is good to meet you, Al'。这个字符串传给print()，它将输出到屏幕上。

1.6.5 len()函数

你可以向len()函数传递一个字符串（或包含字符串的变量），然后该函数求值为一个整型值，即字符串中字符的个数。

```
❺ print('The length of your name is:')
  print(len(myName))
```

在交互式环境中输入以下内容试一试：

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

就像这些例子，len(myName)求值为一个整数。然后它被传递给print()，在屏幕上显示。请注意，print()允许传入一个整型值或字符串。但如果在交互式环境中输入以下内容，就会报错：

```
>>> print('I am ' + 29 + ' years old.')
```

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

导致错误的原因不是print()函数，而是你试图传递给print()的表达式。如果在交互式环境中单独输入这个表达式，也会得到同样的错误。

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
   'I am ' + 29 + ' years old.'
TypeError: Can't convert 'int' object to str implicitly
```

报错是因为，只能用+操作符加两个整数，或连接两个字符串。不能让一个整数和一个字符串相加，因为这不符合Python的语法。可以使用字符串版本的整数，修复这个错误。这在下一节中解释。

1.6.6 str()、int()和float()函数

如果想要连接一个整数（如29）和一个字符串，再传递给print()，就需要获得值'29'。它是29的字符串形式。str()函数可以传入一个整型值，并求值为它的字符串形式，像下面这样：

```
>>> str(29)
```

```
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

因为str(29)求值为’29’，所以表达式'I am ' + str(29) + ' years old.'求值为'I am ' + '29' + ' years old.'，它又求值为'I am 29 years old.'。这就是传递给print()函数的值。

str()、int()和float()函数将分别求值为传入值的字符串、整数和浮点数形式。请尝试用这些函数在交互式环境中转换一些值，看看会发生什么。

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
```

```
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

前面的例子调用了str()、int()和float()函数，向它们传入其他数据类型的值，得到了字符串、整型或浮点型的值。

如果想要将一个整数或浮点数与一个字符串连接，str()函数就很方便。如果你有一些字符串值，希望将它们用于数学运算，int()函数也很有用。例如，input()函数总是返回一个字符串，即便用户输入的是一个数字。在交互式环境中输入spam = input()，在它等待文本时输入101。

```
>>> spam = input()
101
>>> spam
'101'
```

保存在spam中的值不是整数101，而是字符串'101'。如果想要用spam中的值进行数学运算，那就用int()函数取得spam的整数形式，然后将这个新值存在spam中。

```
>>> spam = int(spam)
>>> spam
101
```

现在你应该能将spam变量作为整数，而不是字符串使用。

```
>>> spam * 10 / 5
```

```
202.0
```

请注意，如果你将一个不能求值为整数的值传递给int()，Python将显示出错信息。

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

如果需要对浮点数进行取整运算，也可以用int()函数。

```
>>> int(7.7)
```

```
7
>>> int(7.7) + 1
8
```

在你的程序中，最后3行使用了函数int()和str()，取得适当数据类型的值。

```
❻ print('What is your age?') # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

myAge变量包含了input()函数返回的值。因为input()函数总是返回一个字符串（即使用户输入的是数字），所以你可以使用int(myAge)返回字符串的整型值。这个整型值随后在表达式int(myAge) + 1中与1相加。

相加的结果传递给str()函数：str(int(myAge) + 1)。然后，返回的字符串与字符串'You will be '和' in a year.'连接，求值为一个更长的字符串。这个更长的字符串最终传递给print()，在屏幕上显示。

假定用户输入字符串'4'，保存在myAge中。字符串'4'被转换为一个整型，所以你可以对它加1。结果是5。str()函数将这个结果转化为字符串，这样你就可以将它与第二个字符串'in a year.'连接，创建最终的消息。这些求值步骤如图1-4所示。

文本和数字相等判断

虽然数字的字符串值被认为与整型值和浮点型值完全不同，但整型值可以与浮点值相

等。

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python进行这种区分，因为字符串是文本，而整型值和浮点型都是数字。

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(    4 + 1     ) + ' in a year.')
print('You will be ' + str(      5       ) + ' in a year.')
print('You will be ' +          '5'        + ' in a year.')
print('You will be 5'                      + ' in a year.')
print('You will be 5 in a year.')
```

图1-4 如果4保存在myAge中，求值的步骤

1.7 小结

你可以用一个计算器来计算表达式，或在文本处理器中输入字符串连接。甚至可以通过复制粘贴文本，很容易地实现字符串复制。但是表达式以及组成它们的值（操作符、变量和函数调用），才是构成程序的基本构建块。一旦你知道如何处理这些元素，就能够用Python操作大量的数据。

最好是记住本章中介绍的不同类型的操作符（+、-、*、/、//、%和**是数学操作符，+和*是字符串操作符），以及3种数据类型（整型、浮点型和字符串）。

我们还介绍了几个不同的函数。print()和input()函数处理简单的文本输出（到屏幕）和输入（通过键盘）。len()函数接受一个字符串，并求值为该字符串中字符的数目。

在下一章中，你将学习如何告诉Python根据它拥有的值，明智地决定什么代码要运行，什么代码要跳过，什么代码要重复。这称为“控制流”，它让你编写程序来做出明智的决定。

1.8 习题

1．下面哪些是操作符，哪些是值？

```
*
'hello'
-88.8
-
/
+
5
```

2．下面哪个是变量，哪个是字符串？

```
spam
'spam'
```

3．说出3种数据类型。

4．表达式由什么构成？所有表达式都做什么事？

5．本章介绍了赋值语句，如spam = 10。表达式和语句有什么区别？

6．下列语句运行后，变量bacon的值是什么？

```
bacon = 20
bacon + 1
```

7．下面两个表达式求值的结果是什么？

```
'spam' + 'spamspam'
'spam' * 3
```

8．为什么eggs是有效的变量名，而100是无效的？

9．哪3个函数能分别取得一个值的整型、浮点型或字符串版本？

10．为什么这个表达式会导致错误？如何修复？

```
'I have eaten ' + 99 + ' burritos.'
```

附加题：在线查找len()函数的Python文档。它在一个标题为“Built-in Functions”的网页上。扫一眼Python的其他函数的列表，查看round()函数的功能，在交互式环境中使用它。

第2章 控制流

你已经知道了单条指令的基本知识。程序就是一系列指令。但编程真正的力量不仅在于运行（或“执行”）一条接一条的指令，就像周末的任务清单那样。根据表达式求值的结果，程序可以决定跳过指令，重复指令，或从几条指令中选择一条运行。实际上，你几乎永远不希望程序从第一行代码开始，简单地执行每行代码，直到最后一行。“控制流语句”可以决定在什么条件下执行哪些Python语句。

这些控制流语句直接对应于流程图中的符号，所以在本章中，我将提供示例代码的流程图。图2-1展示了一张流程图，内容是如果下雨怎么办。按照箭头构成的路径，从开始到结束。

在流程图中，通常有不止一种方法从开始走到结束。计算机程序中的代码行也是这样。流程图用菱形表示这些分支节点，其他步骤用矩形表示。开始和结束步骤用带圆角的矩形表示。

但在学习流程控制语句之前，首先要学习如何表示这些yes和no选项。同时你也需要理解，如何将这些分支节点写成Python代码。要做到这一点，让我们先看看布尔值、比较操作符和布尔操作符。

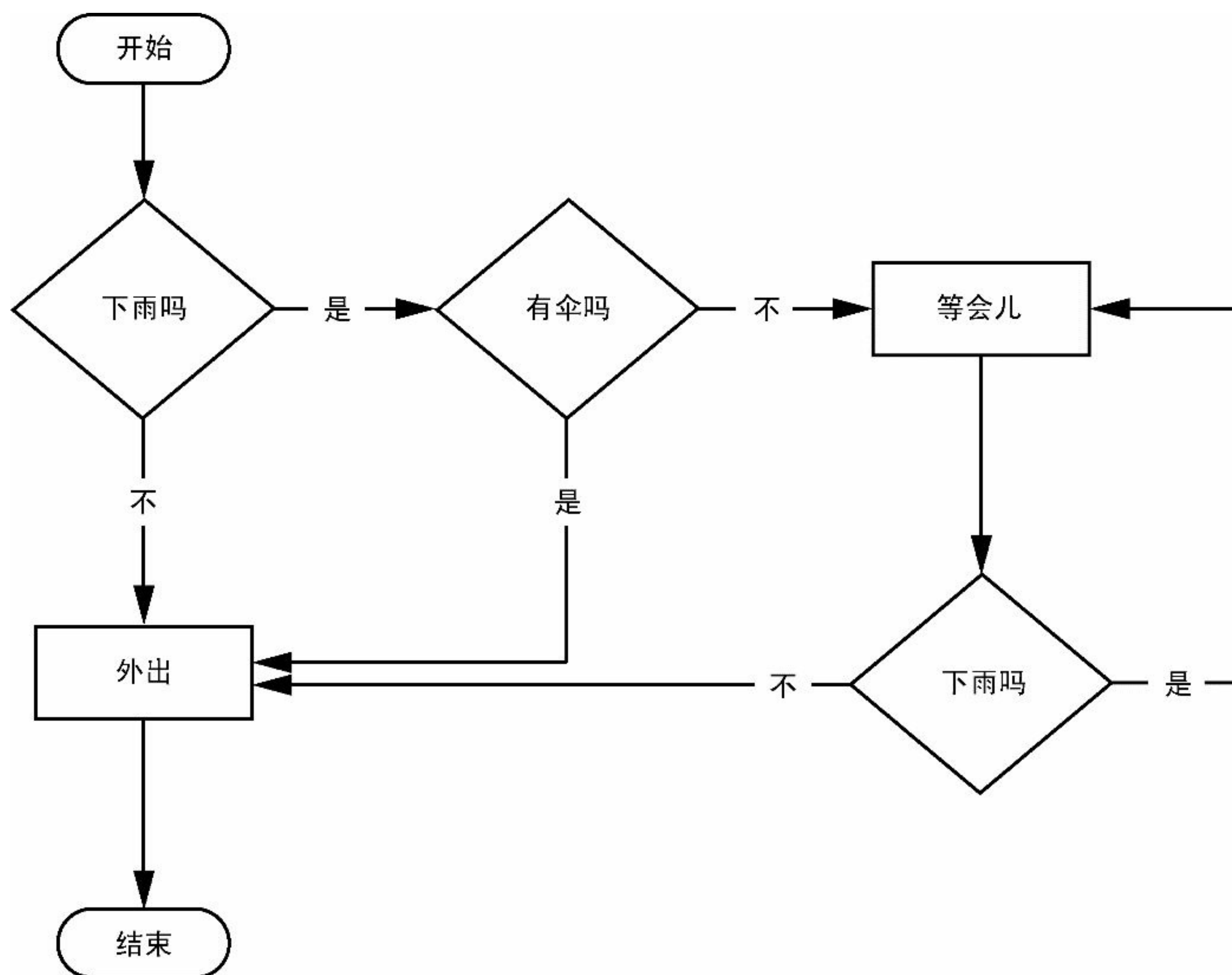


图2-1 一张流程图，告诉你如果下雨要做什么

2.1 布尔值

虽然整型、浮点型和字符串数据类型有无数种可能的值，但“布尔”数据类型只有两种值：True和False。Boolean（布尔）的首字母大写，因为这个数据类型是根据数学家George Boole命名的。在作为Python代码输入时，布尔值True和False不像字符串，两边没有引号，它们总是以大写字母T或F开头，后面的字母小写。在交互式环境中输入下面内容，其中有些指令是故意弄错的，它们将导致出错信息。

```
❶ >>> spam = True
```

```
  >>> spam
  True
❷>>> true
  Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
      true
  NameError: name 'true' is not defined
❸ >>> True = 2 + 2
  SyntaxError: assignment to keyword
```

像其他值一样，布尔值也用在表达式中，并且可以保存在变量中❶。如果大小写不正确❷，或者试图使用True和False作为变量名❸，Python就会给出错误信息。

2.2 比较操作符

“比较操作符”比较两个值，求值为一个布尔值。表2-1列出了比较操作符。

表**2-1** 比较操作符

操作符	含义

==	等于
!=	不等于
<	小于
>	大于
<=	小于等于
>=	大于等于

这些操作符根据给它们提供的值，求值为True或False。现在让我们尝试一些操作符，从==和！=开始。

```
>>>  42 == 42
True
>>>  42 == 99
False
>>>  2 != 3
True
>>>  2 != 2
```

```
False
```

如果两边的值一样，==（等于）求值为True。如果两边的值不同，!=（不等于）求值为True。==和!=操作符实际上可以用于所有数据类型的值。

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
```

```
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

请注意，整型或浮点型的值永远不会与字符串相等。表达式42 == '42'❶求值为False是因为，Python认为整数42与字符串'42'不同。

另一方面，<、>、<=和>=操作符仅用于整型和浮点型值。

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
```

```
  False
  >>> eggCount = 42
❷>>> eggCount <= 42
  True
  >>> myAge = 29
❸ >>> myAge >= 10
  True
```

操作符的区别

你可能已经注意到，==操作符（等于）有两个等号，而=操作符（赋值）只有一个等号。这两个操作符很容易混淆。只要记住：

- ==操作符（等于）问两个值是否彼此相同。
- =操作符（赋值）将右边的值放到左边的变量中。

为了记住谁是谁，请注意==操作符（等于）包含两个字符，就像!=操作符（不等于）包含两个字符一样。

你会经常用比较操作符比较一个变量和另外某个值。就像在例子

eggCount <= 42❶和myAge >= 10❷中一样（毕竟，除了在代码中输入'dog' != 'cat'以外，你本来也可以直接输入True）。稍后，在学习控制流语句时，你会看到更多的例子。

2.3 布尔操作符

3个布尔操作符（and、or和not）用于比较布尔值。像比较操作符一样，它们将这些表达式求值为一个布尔值。让我们仔细看看这些操作符，从and操作符开始。

2.3.1 二元布尔操作符

and和or操作符总是接受两个布尔值（或表达式），所以它们被认为是“二元”操作符。如果两个布尔值都为True，and操作符就将表达式求值为True，否则求值为False。在交互式环境中输入某个使用and的表达式，看看效果。

```
>>> True and True
True
>>> True and False
False
```

“真值表”显示了布尔操作符的所有可能结果。表2-2是操作符and的真值表。

表**2-2 and**操作符的真值表

表达式	求值为
True and True	True
True and False	False
False and True	False
False and False	False

另一方面，只要有一个布尔值为真，or操作符就将表达式求值为True。如果都是False，所求值为False。

```
>>> False or True
True
>>> False or False
False
```

可以在or操作符的真值表中看到每一种可能的结果，如表2-3所示。

表**2-3 or**操作符的真值表

表达式	求值为

True or True	True
True or False	True
False or True	True
False or False	False

2.3.2 not操作符

和and和or不同，not操作符只作用于一个布尔值（或表达式）。not操作符求值为相反的布尔值。

```
  >>> not   True
  False
❶ >>> not not not not True
  True
```

就像在说话和写作中使用双重否定，你可以嵌套not操作符❶，虽然在真正的程序中并不经常这样做。表2-4展示了not的真值表。

表**2-4 not**操作符的真值表

表达式	求值为
not True	False
not False	True

2.4 混合布尔和比较操作符

既然比较操作符求值为布尔值，就可以和布尔操作符一起，在表达式中使用。

回忆一下，and、or和not操作符称为布尔操作符是因为，它们总是操作于布尔值。虽然像4 < 5这样的表达式不是布尔值，但可以求值为布尔值。在交互式环境中，尝试输入一些使用比较操作符的布尔表达式。

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

计算机将先求值左边的表达式，然后再求值右边的表达式。知道两个布尔值后，它又将整个表达式再求值为一个布尔值。你可以认为计算机求值(4 < 5)和(5 < 6)的过程，如图2-2所示。

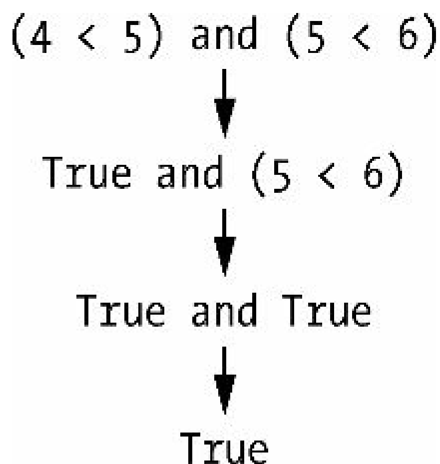


图2-2 (4 < 5)和 (5 < 6) 求值为True的过程

也可以在一个表达式中使用多个布尔操作符，与比较操作符一起使用。

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

和算术操作符一样，布尔操作符也有操作顺序。在所有算术和比较操作符求值后，Python先求值not操作符，然后是and操作符，然后是or操作符。

2.5 控制流的元素

控制流语句的开始部分通常是“条件”，接下来是一个代码块，称为“子句”。在开始学习具体的Python控制流语句之前，我将介绍条件和代码块。

2.5.1 条件

你前面看到的布尔表达式可以看成是条件，它和表达式是一回事。“条件”只是在控制流语句的上下文中更具体的名称。条件总是求值为一个布尔值，True或False。控制流语句根据条件是True还是False，来决定做什么。几乎所有的控制流语句都使用条件。

2.5.2 代码块

一些代码行可以作为一组，放在“代码块”中。可以根据代码行的缩进，知道代码块的开始和结束。代码块有3条规则。

1．缩进增加时，代码块开始。

2．代码块可以包含其他代码块。

3．缩进减少为零，或减少为外面包围代码块的缩进，代码块就结束了。

看一些有缩进的代码，更容易理解代码块。所以让我们在一小段游戏程序中，寻找代码块，如下所示：

```
  if name == 'Mary':
❶     print('Hello Mary')
  if password == 'swordfish':
❷    print('Access granted.')
  else:
❸     print('Wrong password.')
```

第一个代码块❶开始于代码行print('Hello Mary')，并且包含后面所有的行。在这个代码块中有另一个代码块❷，它只有一行代码：

print('Access Granted.')。第三个代码块❸也只有一行：print('Wrong password.')。

2.6 程序执行

在第1章的hello.py程序中，Python开始执行程序顶部的指令，然后一条接一条往下执行。“程序执行”（或简称“执行”）这一术语是指当前被执行的指令。如果将源代码打印在纸上，在它执行时用手指指着每一行代码，你可以认为手指就是程序执行。

但是，并非所有的程序都是从上至下简单地执行。如果用手指追踪一个带有控制流语句的程序，可能会发现手指会根据条件跳过源代码，有可能跳过整个子句。

2.7 控制流语句

现在，让我们来看最重要的控制流部分：语句本身。语句代表了在图2-1的流程图中看到的菱形，它们是程序将做出的实际决定。

2.7.1 if语句

最常见的控制流语句是if语句。if语句的子句（也就是紧跟if语句的语句块），将在语句的条件为True时执行。如果条件为False，子句将跳过。

在英文中，if语句念起来可能是：“如果条件为真，执行子句中的代码。”在Python中，if语句包含以下部分：

- if关键字；
- 条件（即求值为True或False的表达式）；
- 冒号；
- 在下一行开始，缩进的代码块（称为if子句）。

例如，假定有一些代码，检查某人的名字是否为Alice（假设此前曾为name赋值）。

```
if name == 'Alice':
```

```
    print('Hi, Alice.')
```

所有控制流语句都以冒号结尾，后面跟着一个新的代码块（子句）。语句的if子句是代码块，包含print('Hi, Alice.')。图2-3展示了这段代码的流程图。

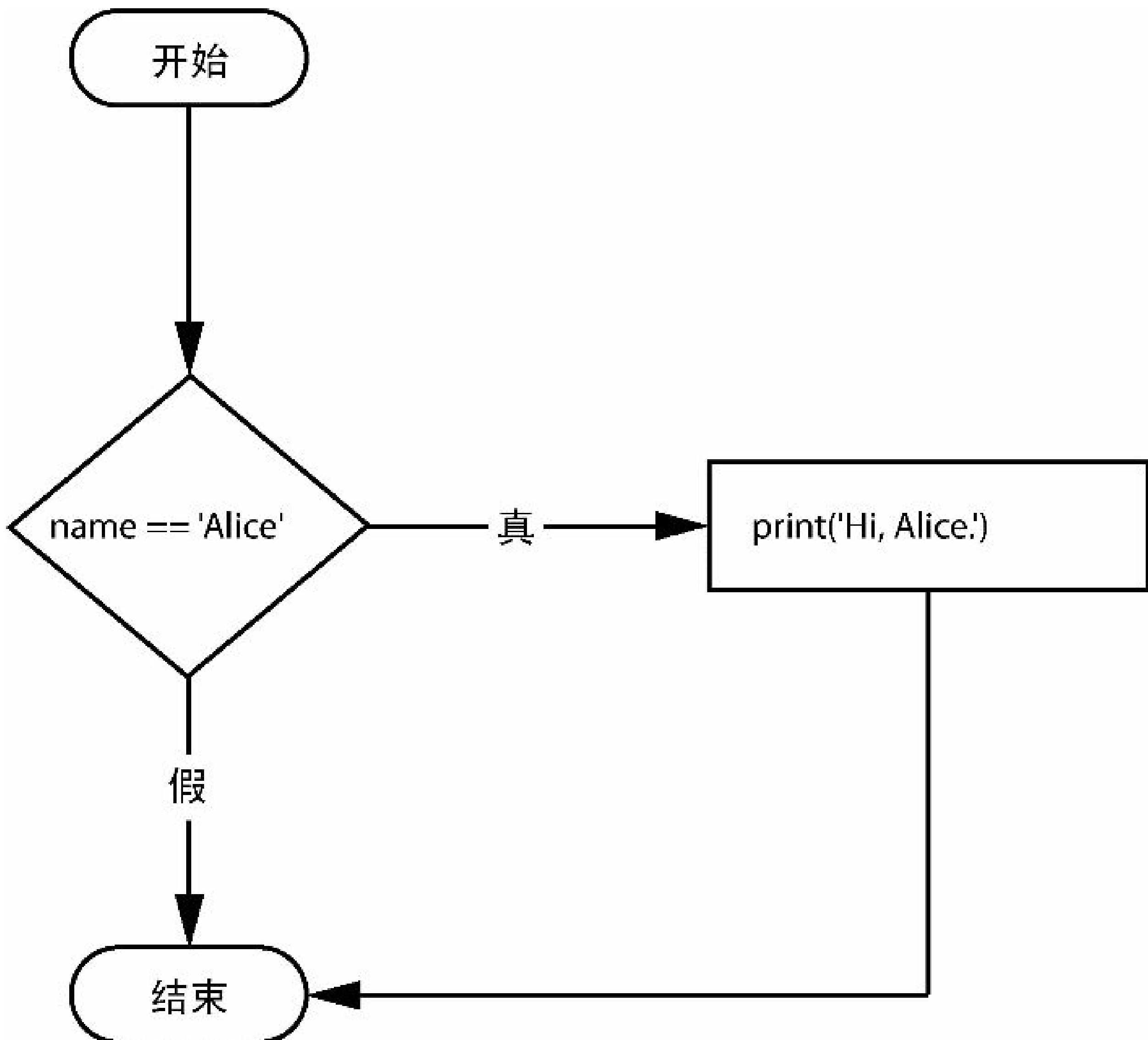


图2-3 if语句的流程图

2.7.2 else语句

if子句后面有时候也可以跟着else语句。只有if语句的条件为False时，else子句才会执行。在英语中，else语句读起来可能是："如果条件为真，执行这段代码。否则，执行那段代码"。else语句不包含条件，在代码中，else语句中包含下面部分：

- else关键字；
- 冒号；
- 在下一行开始，缩进的代码块（称为else子句）。

回到Alice的例子，我们来看看使用else语句的一些代码，在名字不是Alice时，提供不一样的问候。

```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

图2-4展示了这段代码的流程图。

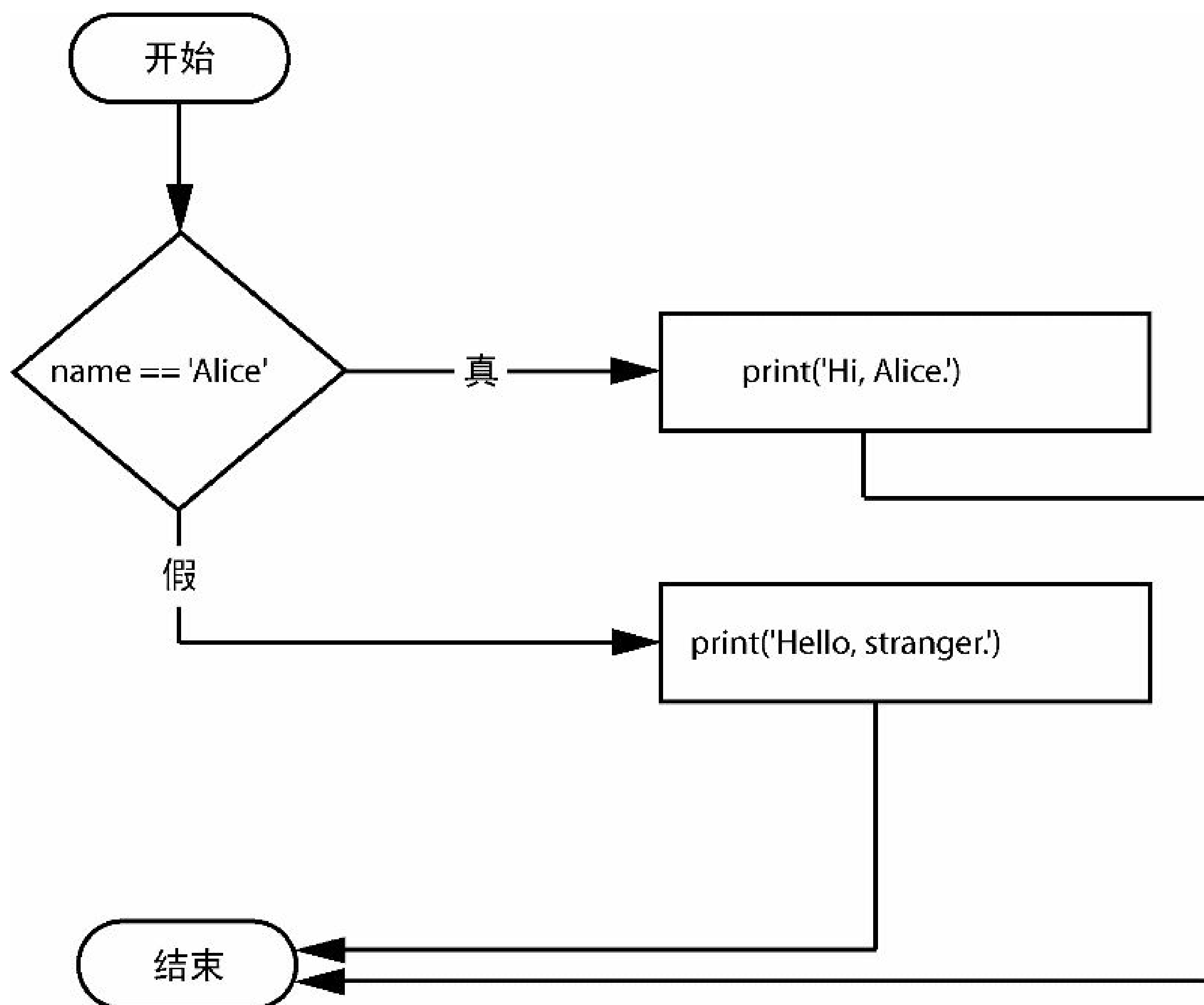


图2-4 else语句的流程图

2.7.3 elif语句

虽然只有if或else子句会被执行，但有时候可能你希望，“许多”可能的子句中有一个被执行。elif语句是“否则如果”，总是跟在if或另一条elif语句后面。它提供了另一个条件，仅在前面的条件为False时才检查该条件。在代码中，elif语句总是包含以下部分：

- elif关键字；
- 条件（即求值为True或False的表达式）；
- 冒号；
- 在下一行开始，缩进的代码块（称为elif子句）。

让我们在名字检查程序中添加elif，看看这个语句的效果。

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

这一次，检查此人的年龄。如果比 12 岁小，就告诉他一些不同的东西。可以在图2-5中看到这段代码的流程图。

如果age < 12为True并且name == 'Alice'为False，elif子句就会执行。但是，如果两个条件都为False，那么两个子句都会跳过。“不能”保证至少有一个子句会被执行。如果有一系列的elif语句，仅有一条或零条子句会被执行。一旦一个语句的条件为True，剩下的elif子句会自动跳过。例如，打开一个新的文件编辑器窗口，输入以下代码，保存为vampire.py。

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

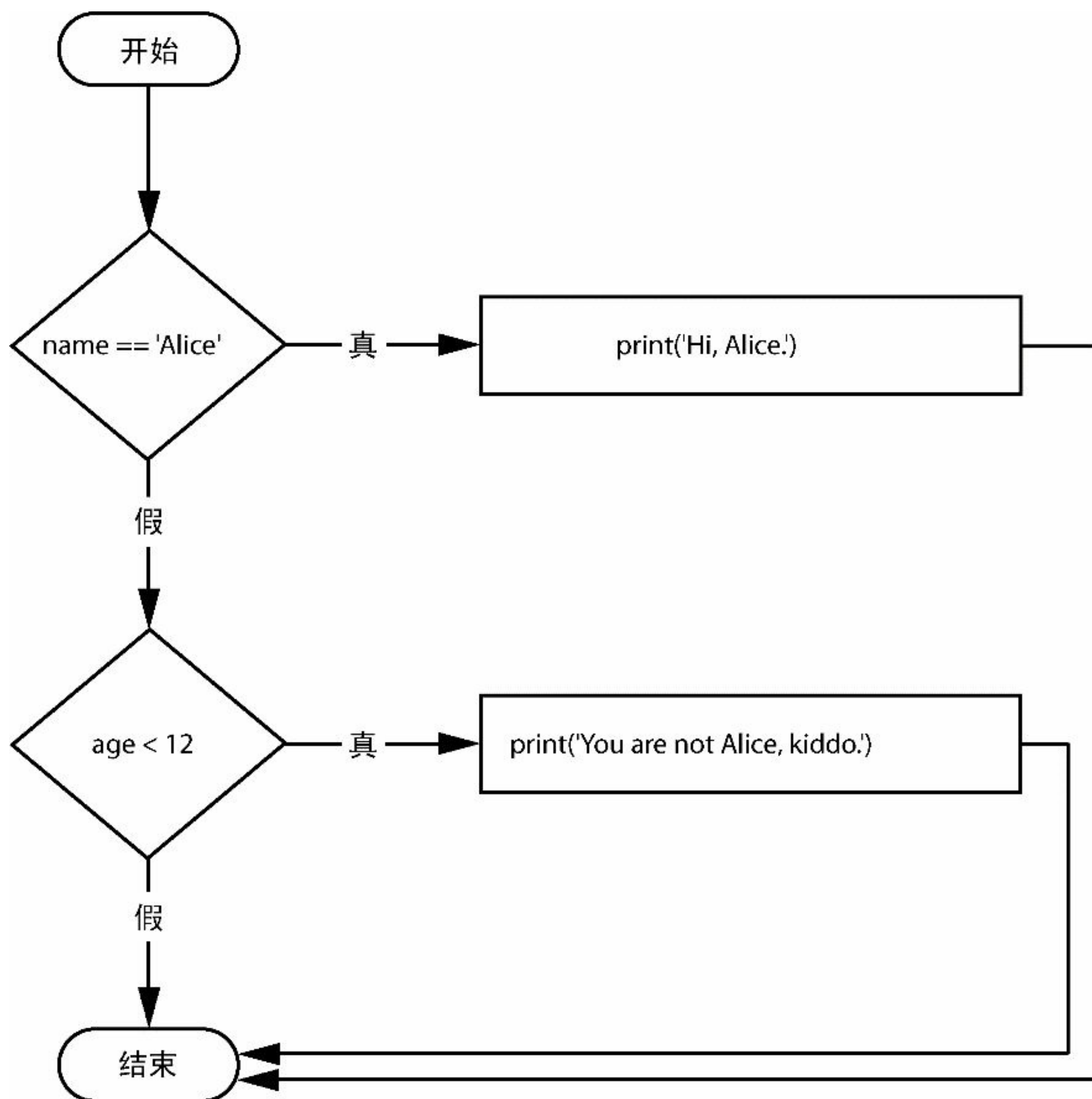


图2-5 elif语句的流程图

这里，我添加了另外两条elif语句，让名字检查程序根据age的不同答案而发出问候。图2-6展示了这段代码的流程图。

但是，elif语句的次序确实重要。让我们重新排序，引入一个缺陷。回忆一下，一旦找到一个True条件，剩余的子句就会自动跳过。所以如果交换vampire.py中的一些子句，就会遇到问题。像下面这样改变代码，将它保存为vampire2.py。

```
  if name == 'Alice':
      print('Hi, Alice.')
  elif age < 12:
      print('You are not Alice, kiddo.')
❶ elif age > 100:
      print('You are not Alice, grannie.')
  elif age > 2000:
      print('Unlike you, Alice is not an undead, immortal vampire.')
```

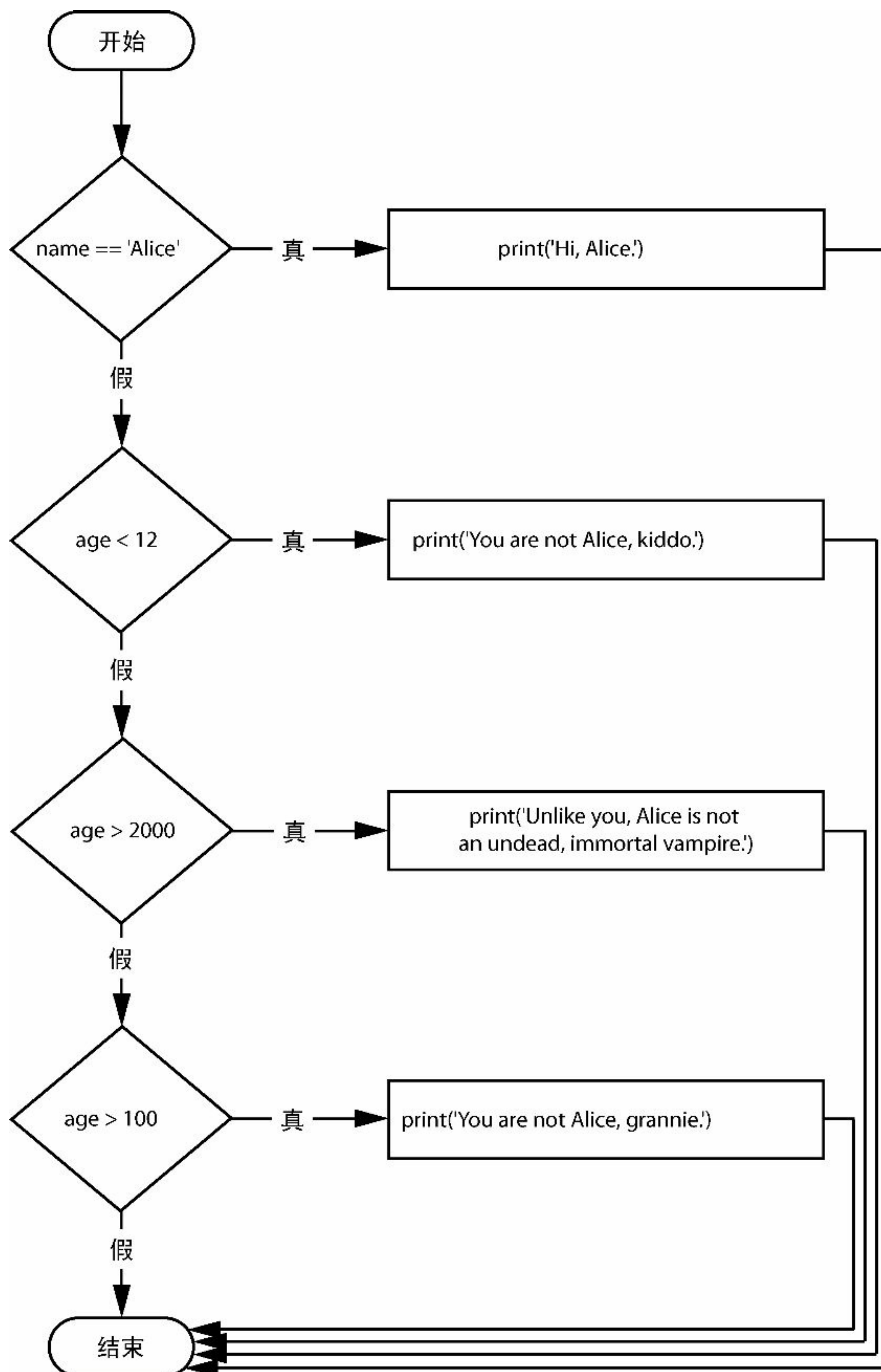
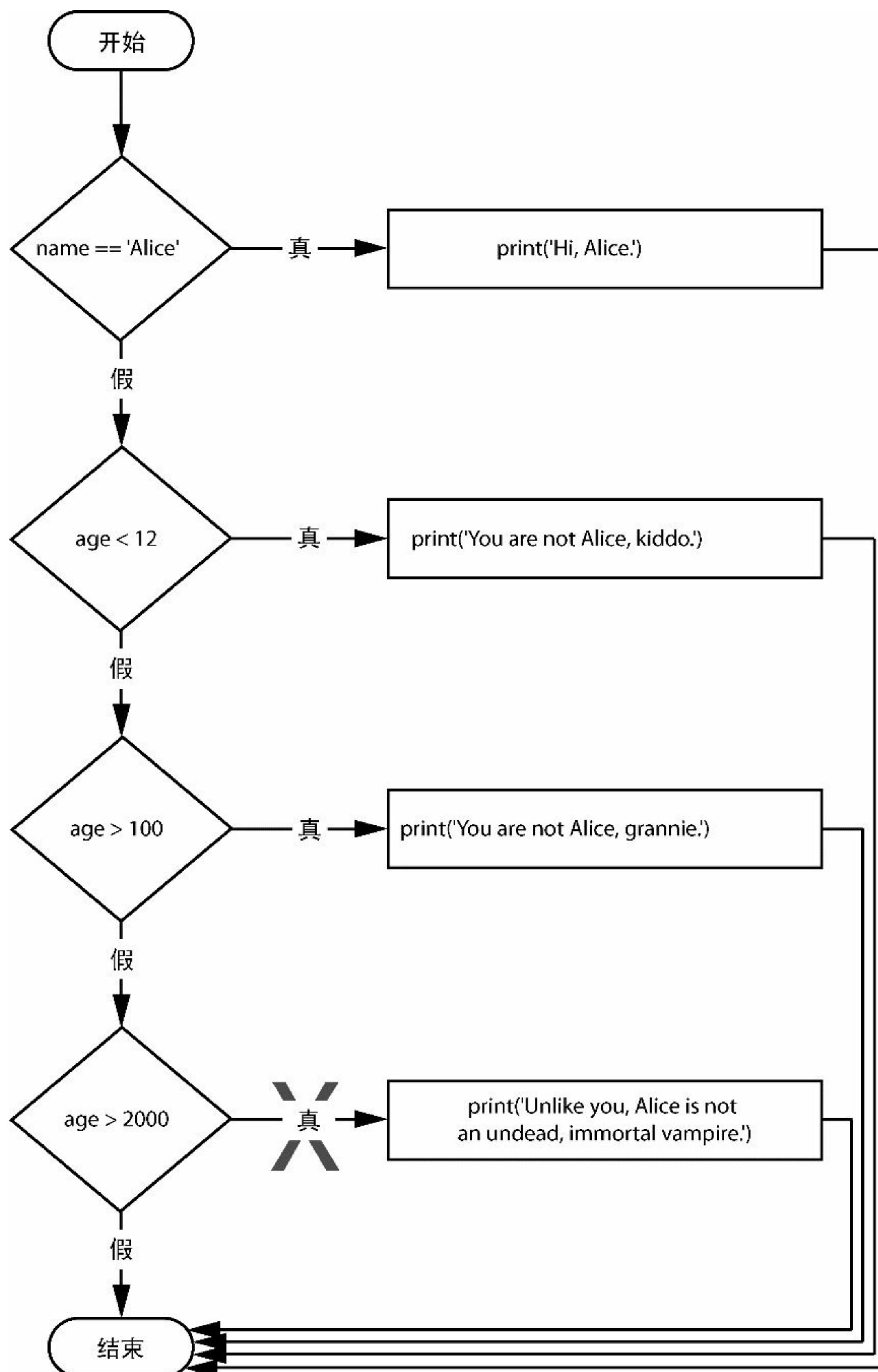

开始
name == 'Alice'
真
print('Hi, Alice.')
假
age < 12
真
print('You are not Alice, kiddo.')
假
age > 2000
真
print('Unlike you, Alice is not
an undead, immortal vampire.')
假
age > 100
真
print('You are not Alice, grannie.')
假
结束

图2-6 vampire.py程序中多重elif语句的流程图

假设在这段代码执行之前，age变量的值是3000。你可能预计代码会打印出字符串'Unlike you, Alice is not an undead, immortal vampire.'。但是，因为age > 100条件为真（毕竟3000大于100）❶，字符串'You are not Alice, grannie.'被打印出来，剩下的语句自动跳过。别忘了，最多只有一个子句会执行，对于elif语句，次序是很重要的。

图2-7展示了前面代码的流程图。请注意，菱形age > 100和age > 2000交换了位置。



开始
name == 'Alice'
真
print('Hi, Alice.')
假
age < 12
真
print('You are not Alice, kiddo.')
假
age > 100
真
print('You are not Alice, grannie.')
假
age > 2000
真
print('Unlike you, Alice is not
an undead, immortal vampire.')
假
结束

图2-7 vampire2.py程序的流程图。打叉的路径在逻辑上永远不会发生，
因为如果age大于2000，它就已经大于100了

你可以选择在最后的elif语句后面加上else语句。在这种情况下，保证至少一个子句（且只有一个）会执行。如果每个if和elif语句中的条件都为False，就执行else子句。例如，让我们使用if、elif和else子句重新编写Alicee程序。

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

图2-8展示了这段新代码的流程图，我们将它保存为littleKid.py。

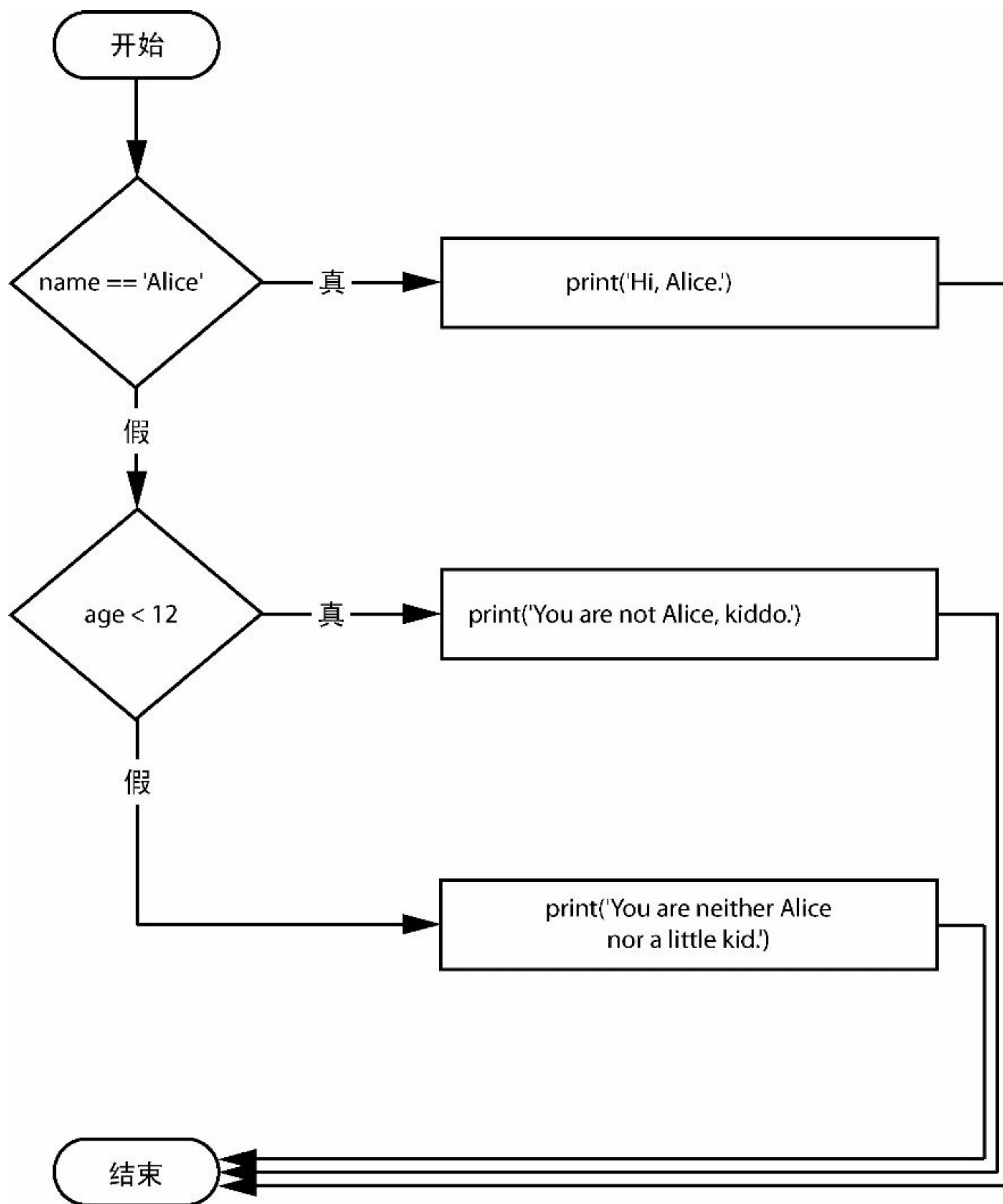


图2-8 前面littleKid.py程序的流程图

在英语中，这类控制流结构会使得：“如果第一个条件为真，做这个。否则，如果第二个条件为真，做那个。否则，做另外的事。”如果

你同时使用这3个语句，要记住这些次序规则，避免图2-7中那样的缺陷。首先，总是只有一个if语句。所有需要的elif语句都应该跟在if语句之后。其次，如果希望确保至少一条子句被执行，在最后加上else语句。

2.7.4 while循环语句

利用while语句，可以让一个代码块一遍又一遍的执行。只要while语句的条件为True，while子句中的代码就会执行。在代码中，while语句总是包含下面几部分：

- 关键字；
- 条件（求值为True或False的表达式）；
- 冒号；
- 从新行开始，缩进的代码块（称为while子句）。

可以看到，while语句看起来和if语句类似。不同之处是它们的行为。if子句结束时，程序继续执行if语句之后的语句。但在while子句结束时，程序执行跳回到while语句开始处。while子句常被称为“while循环”，或就是“循环”。

让我们来看一个if语句和一个while循环。它们使用同样的条件，并基于该条件做出同样的动作。下面是if语句的代码：

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

下面是while语句的代码：

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

这些语句类似，if和while都检查spam的值，如果它小于 5 ，就打印一条消息。但如果运行这两段代码，它们各自的表现非常不同。对于if语句，输出就是"Hello, world."。但对于while语句，输出是"Hello, world."重复了5次！看一看这两段代码的流程图，图2-9和2-10，找一找原因。

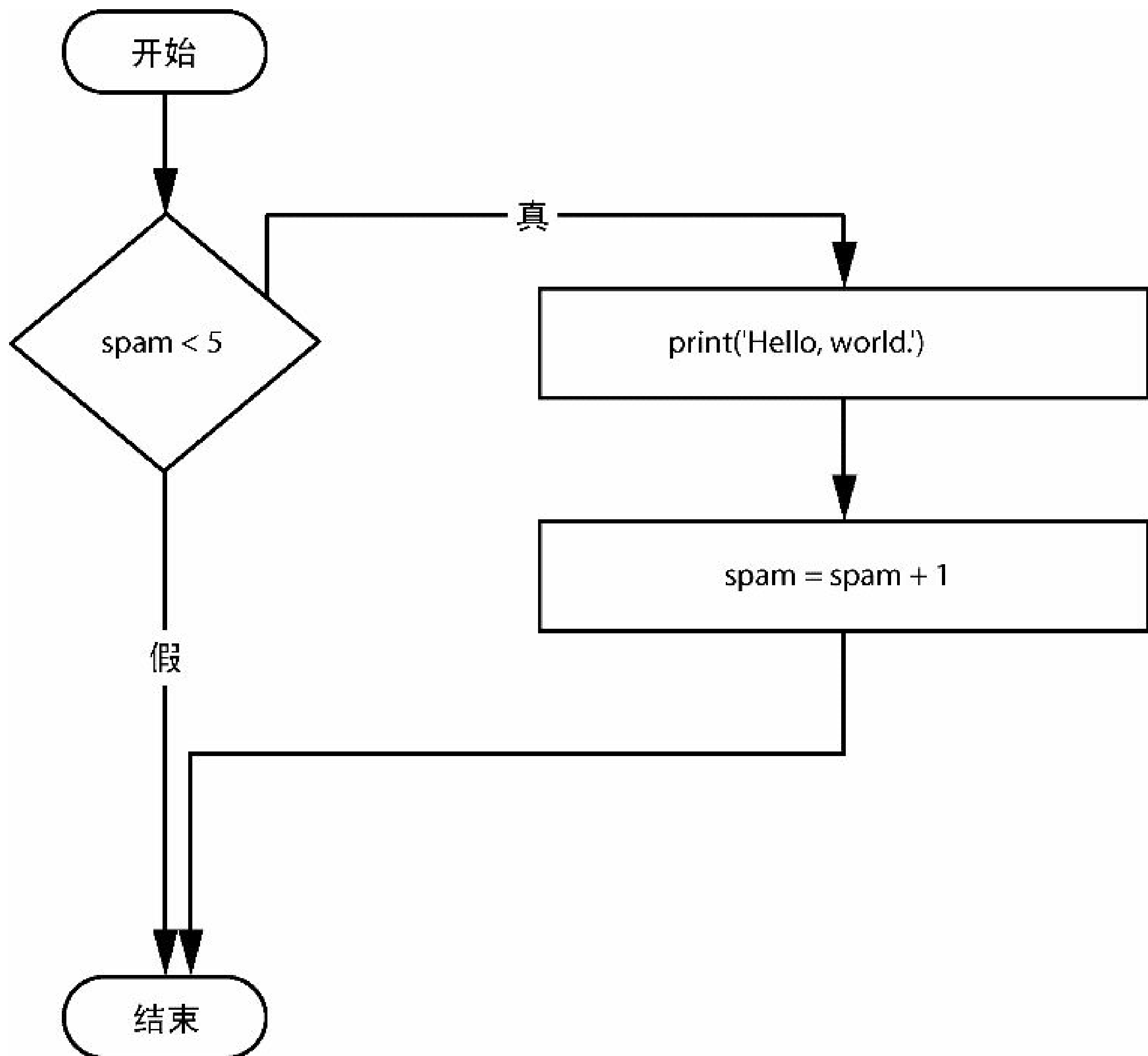


图2-9 if语句代码的流程图

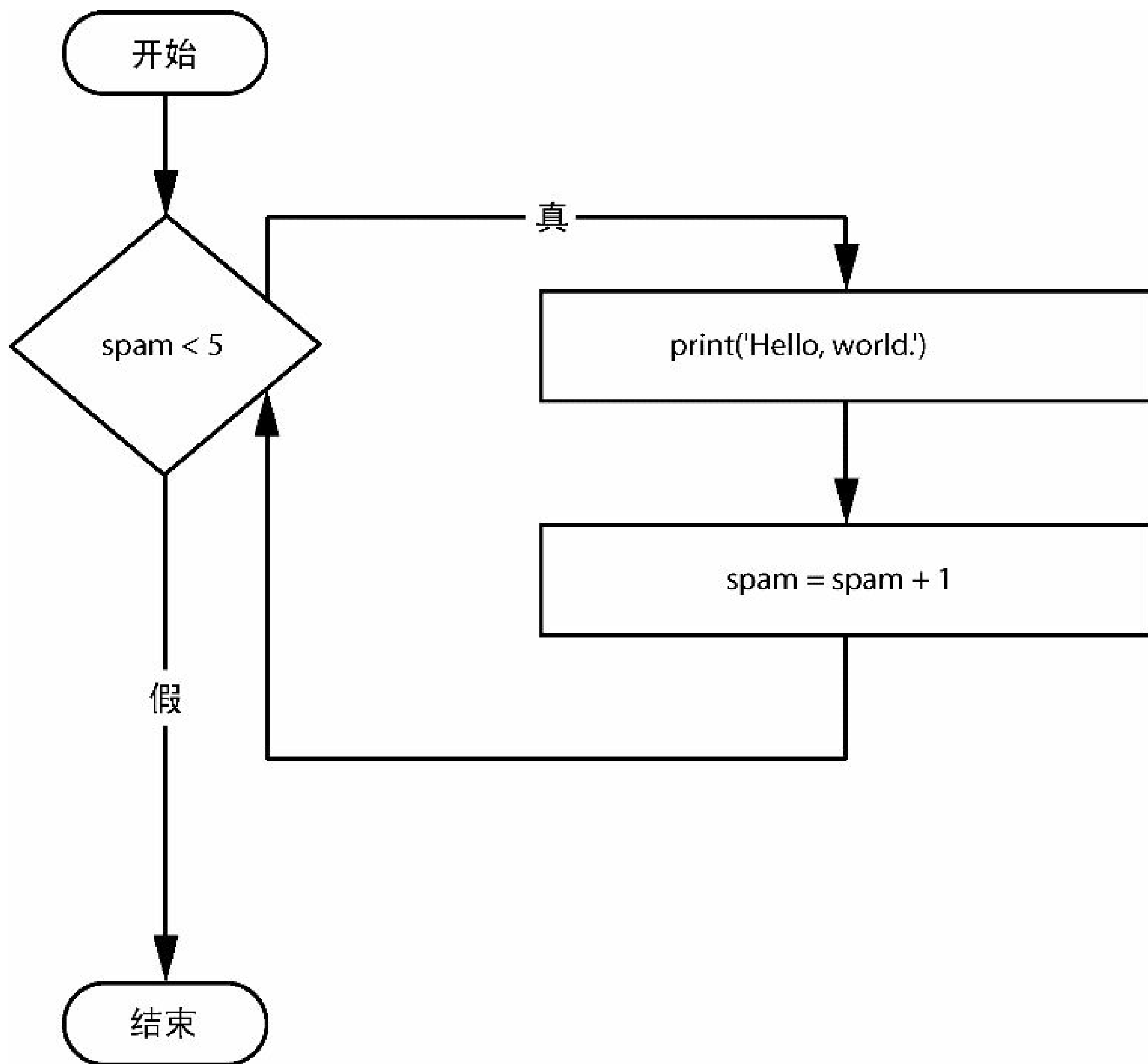


图2-10 while语句代码的流程图

带有if语句的代码检查条件，如果条件为True，就打印一次"Hello, world."。带有while循环的代码则不同，会打印5次。打印5次后停下来是因为，在每次循环迭代末尾，spam中的整数都增加1。这意味着循环将执行5次，然后spam < 5变为False。

在while循环中，条件总是在每次“迭代”开始时检查（也就是每次循环执行时）。如果条件为True，子句就会执行，然后，再次检查条件。当条件第一次为False时，while子句就跳过。

2.7.5 恼人的循环

这里有一个小例子，它不停地要求你输入“your name”（就是这个字符串，而不是你的名字）。选择FileŔNew Window，打开一个新的文件编辑器窗口，输入以下代码，将文件保存为yourName.py：

```
❶ name = ''
❷while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

首先，程序将变量name❶设置为一个空字符串。这样，条件name != 'your name'就会求值为True，程序就会进入while循环的子句❷。

这个子句内的代码要求用户输入他们的名字，然后赋给name变量❸。因为这是语句块的最后一行，所以执行就回到while循环的开始，重新对条件求值。如果name中的值“不等于”字符串'your name'，那么条件就为True，执行将再次进入while子句。

但如果用户输入your name，while循环的条件就变成'your name' != 'your name'，它求值为False。条件现在是False，程序就不会再次进入while循环子句，而是跳过它，继续执行程序后面的部分❹。图2-11展示了yourName.py程序的流程图。

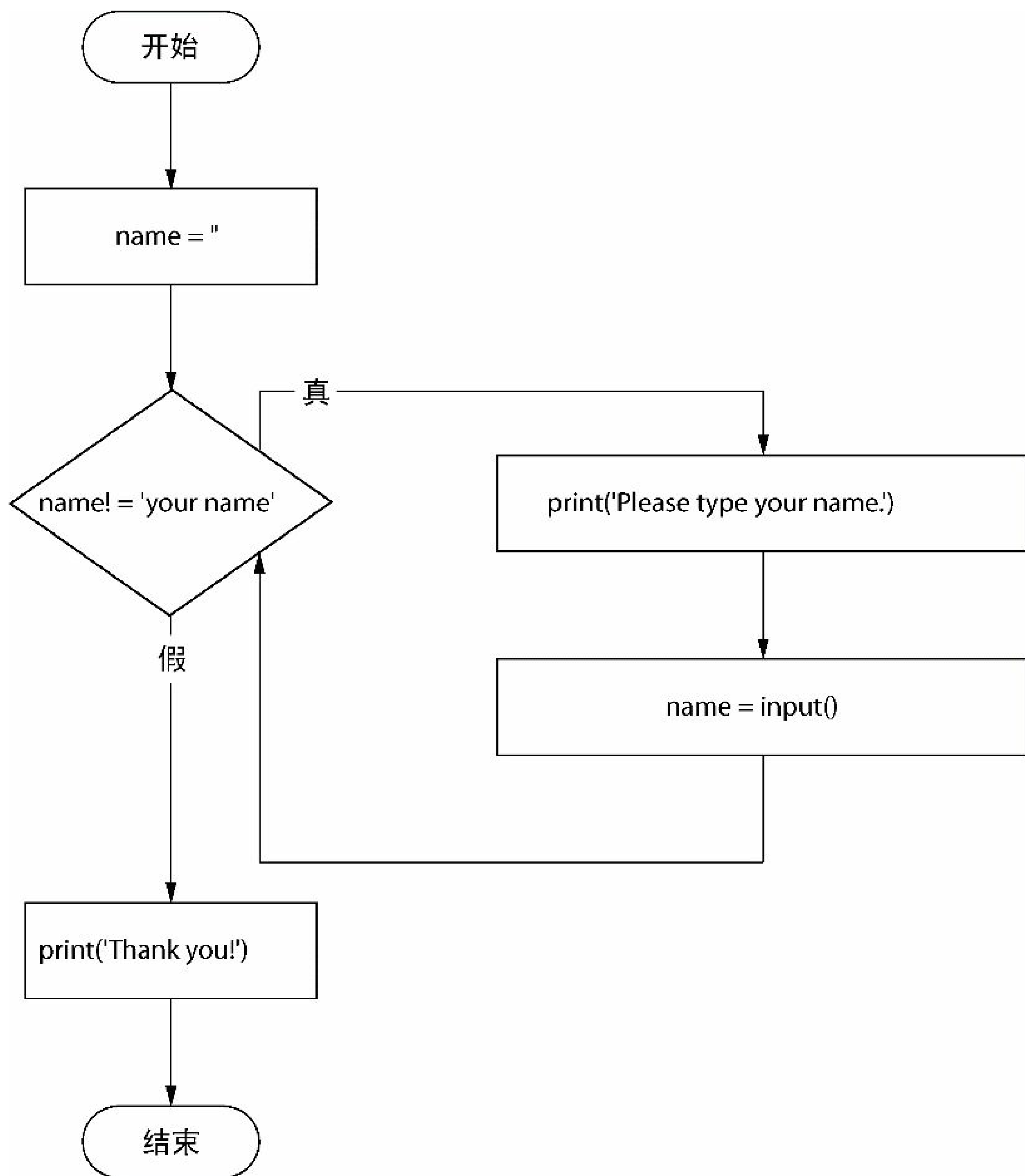


图2-11 yourName.py程序的流程图

现在，让我们来看看yourName.py程序的效果。按F5键运行它，输几次your name之外的东西，然后再提供程序想要的输入。

```
Please type your name.
```

```
Al

Please type your name.
Albert

Please type your name.
%#@#%*(^&!!!

Please type your name.
your name

Thank you!
```

如果永不输入your name，那么循环的条件就永远为False，程序将永远问下去。这里，input()调用让用户输入正确的字符串，以便让程序继续。在其他程序，条件可能永远没有实际变化，这可能会出问题。让我们来看看如何跳出循环。

2.7.6 break语句

有一个捷径，让执行提前跳出while循环子句。如果执行遇到break语句，就会马上退出while循环子句。在代码中，break语句仅包含break关键字。

非常简单，对吗？这里有一个程序，和前面的程序做一样的事情，但使用了break语句来跳出循环。输入以下代码，将文件保存为yourName2.py：

```
❶ while True:
     print('Please type your name.')
❷    name = input()
❸    if name == 'your name':
❹        break
❺ print('Thank you!')
```

第一行❶创建了一个“无限循环”，它是一个条件总是为True的while循环。（表达式True总是求值为True。）程序执行将总是进入循环，只有遇到break语句执行时才会退出（“永远不”退出的无限循环是一个常见的编程缺陷）。

像以前一样，程序要求用户输入your name❷。但是现在，虽然执行仍然在while循环内，但有一个if语句会被执行❸，检查name是否等于your name。如果条件为True，break语句就会运行❹，执行就会跳出循环，转到print('Thank you!') ❺。否则，包含break语句的if语句子句就会跳过，让执行到达while循环的末尾。此时，程序执行跳回到while语句的开始❶，重新检查条件。因为条件是True，所以执行进入循环，再次要求用户输入your name。这个程序的流程图参见图2-12。

运行yourName2.py，输入你为yourName.py程序输入的同样文本。重写的程序应该和原来的程序反应相同。

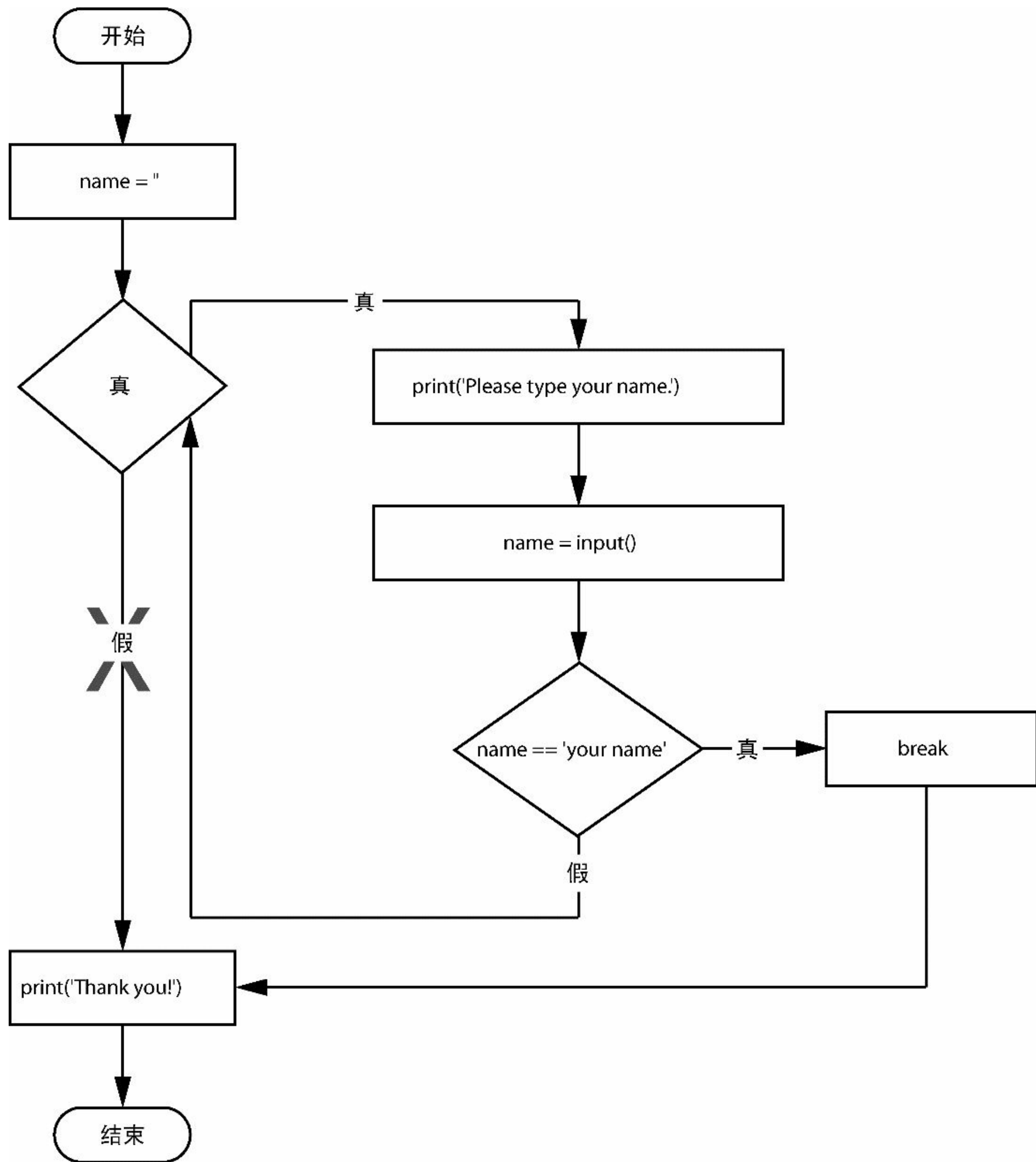


图2-12 带有无限循环的程序的流程图。注意打叉路径在逻辑上永远不会发生，因为循环条件总是为True

2.7.7 continue语句

像break语句一样，continue语句用于循环内部。如果程序执行遇到

continue语句，就会马上跳回到循环开始处，重新对循环条件求值（这也是执行到达循环末尾时发生的事情）。

让我们用continue写一个程序，要求输入名字和口令。在一个新的文件编辑窗口中输入以下代码，将程序保存为swordfish.py。

```
 while True:
     print('Who are you?')
     name = input()
❶    if name != 'Joe':
❷        continue
     print('Hello, Joe. What is the password? (It is a fish.)')
❸    password = input()
     if password == 'swordfish':
❹        break
❺ print('Access granted.')
```

如果用户输入的名字不是Joe❶，continue语句❷将导致程序执行跳回到循环开始处。再次对条件求值时，执行总是进入循环，因为条件就是True。如果执行通过了if语句，用户就被要求输入口令❸。如果输入的口令是swordfish，break语句运行❹，执行跳出while循环，打印Access granted❺。否则，执行继续到while循环的末尾，又跳回到循环的开始。这个程序的流程图参见图2-13。

陷在无限循环中？

如果你运行一个有缺陷的程序，导致陷在一个无限循环中，那么请按Ctrl-C。这将向程序发送KeyboardInterrupt错误，导致它立即停止。试一下，在文件编辑器中创建一个简单的无限循环，将它保存为infiniteloop.py。

```
while True:
   print('Hello world!')
```

如果运行这个程序，它将永远在屏幕上打印Hello world！因为while语句的条件总是True。在IDLE的交互式环境窗口中，只有两种办法停止这个程序：按下Ctrl-C或从菜单中选择ShellṔRestart Shell。如果你希望马上停止程序，即使它不是陷在一个无限循环中，Ctrl-C也是很方便的。

运行这个程序，提供一些输入。只有你声称是Joe，它才会要求输入口令。一旦输入了正确的口令，它就会退出。

```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

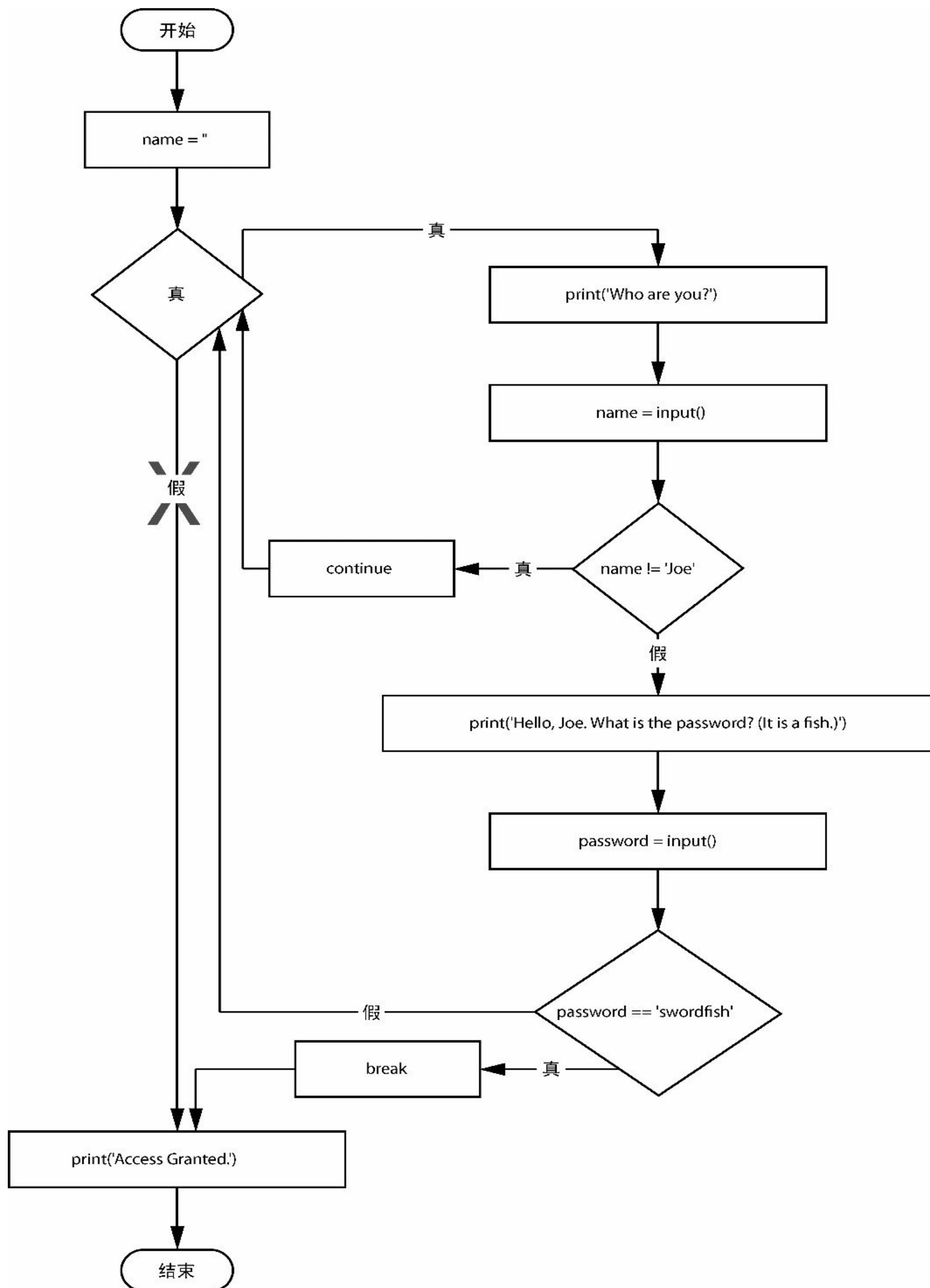

开始
name = ''
真
真
print('Who are you?')
name = input()
假
continue
真
name != 'Joe'
假
print('Hello, Joe. What is the password? (It is a fish.)')
password = input()
假
password == 'swordfish'
break
真
print('Access Granted.')
结束

图2-13 swordfish.py的流程图。打叉的路径在逻辑上永远不会执行，因为循环条件总是True

2.7.8 for循环和range()函数

在条件为True时，while循环就会继续循环（这是它的名称的由来）。但如果你想让一个代码块执行固定次数，该怎么办？可以通过for循环语句和range()函数来实现。

“类真”和“类假”的值

其他数据类型中的某些值，条件认为它们等价于True和False。在用于条件时，0、0.0和''（空字符串）被认为是False，其他值被认为是True。例如，请看下面的程序：

```
name = ''
while not name:❶
print('Enter your name:')
name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests:❷
print('Be sure to have enough room for all your guests.')❸
print('Done')
```

如果用户输入一个空字符串给name，那么while语句的条件就会是True ❶，程序继续要求输入名字。如果numOfGuests不是0 ❷，那么条件就被认为是True，程序就会为用户打印一条提醒信息❸。

可以用not name != ''代替not name，用numOfGuests != 0代替numOfGuests，但使用类真和类假的值会让代码更容易阅读。

在代码中，for语句看起来像for i in range(5):这样，总是包含以下部分：

- for关键字；
- 一个变量名；
- in关键字；
- 调用range()方法，最多传入3个参数；
- 冒号；
- 从下一行开始，缩退的代码块（称为for子句）。

让我们创建一个新的程序，名为fiveTimes.py，看看for循环的效

果。

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

for循环子句中的代码运行了5次。第一次运行时，变量i被设为0。子句中的print()调用将打印出Jimmy Five Times (0)。Python完成for循环子句内所有代码的一次迭代之后，执行将回到循环的顶部，for语句让i增加1。这就是为什么range(5)导致子句的5次迭代，i分别被设置为0、1、2、3、4。变量i将递增到（但不包括）传递给range()函数的整数。图2-14展示了fiveTimes.py程序的流程图。

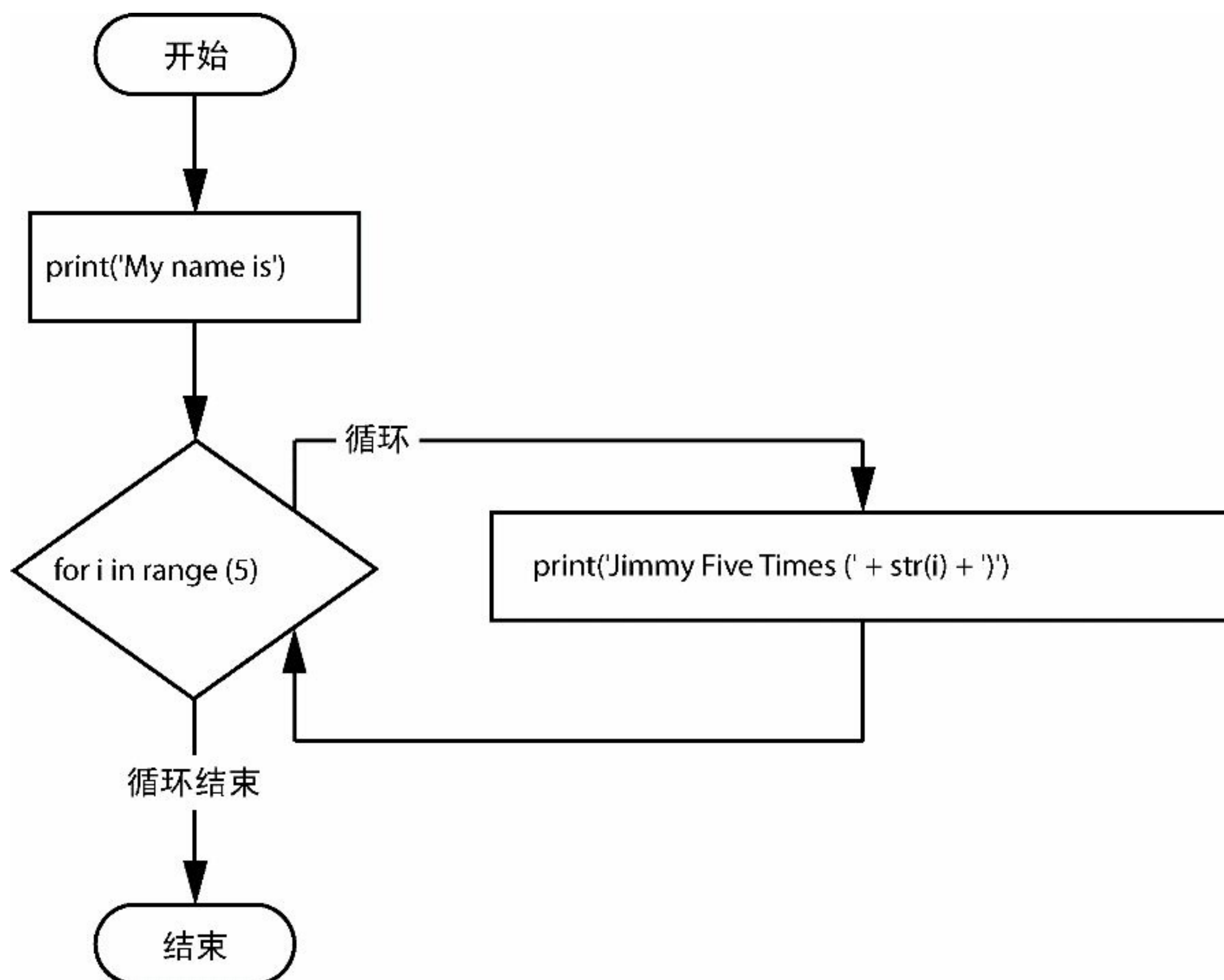


图2-14 fiveTimes.py的流程图

运行这个程序时，它将打印5次Jimmy Five Times和i的值，然后离开for循环。

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

也可以在循环中使用continue语句。continue语句将让for循环变量继续下一个值，就像程序执行已经到达循环的末尾并返回开始一样。实际上，只能在while和for循环内部使用continue和break语句。如果试图在别处使用这些语句，Python将报错。

作为for循环的另一个例子，请考虑数学家高斯的故事。当高斯还是一个小孩时，老师想给全班同学布置很多计算作业。老师让他们从0加到100。高斯想到了一个聪明办法，在几秒钟内算出了答案，但你可以用for循环写一个Python程序，替你完成计算。

```
❶ total = 0
❷for num in range(101):
❸     total = total + num
❹ print(total)
```

结果应该是5050。程序刚开始时，total变量被设为0。然后for循环执行100次total = total + num。当循环完成100次迭代时，0到100的每个整数都加给了total。这时，total被打印到屏幕上。即使在最慢的计算机上，这个程序也不用1秒钟就能完成计算。

（小高斯想到，有50对数加起来是100：1 + 99, 2 + 98, 3 + 97……直到49 + 51。因为50 × 100 是5000，再加上中间的50，所以0到100的所有数之和是5050。聪明的孩子！）

2.7.9 等价的while循环

实际上可以用while循环来做和for循环同样的事，for循环只是更简洁。让我们用与for循环等价的while循环，重写fiveTimes.py。

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

运行这个程序，输出应该和使用for循环的fiveTimes.py程序一样。

2.7.10 range()的开始、停止和步长参数

某些函数可以用多个参数调用，参数之间用逗号分开，range()就是其中之一。这让你能够改变传递给range()的整数，实现各种整数序列，包括从0以外的值开始。

```
for i in range(12, 16):
    print(i)
```

第一个参数是for循环变量开始的值，第二个参数是上限，但不包含它，也就是循环停止的数字。

```
12
13
14
15
```

```
```

range()函数也可以有第三个参数。前两个参数分别是起始值和终止值，第三个参数是“步长”。步长是每次迭代后循环变量增加的值。

```
for i in range(0, 10, 2):
    print(i)
```

所以调用range(0, 10, 2)将从0数到8，间隔为2。

```
0
2
4
6
8
```

在为for循环生成序列数据方面，range()函数很灵活。举例来说，甚至可以用负数作为步长参数，让循环计数逐渐减少，而不是增加。

```
for i in range(5, -1, -1):
    print(i)
```

运行一个for循环，用range(5, -1, -1)来打印i，结果将从5降至0。

```
5
4
3
```

```
2
1
0
```

2.8 导入模块

Python程序可以调用一组基本的函数，这称为“内建函数”，包括你见到过的print()、input()和len()函数。Python也包括一组模块，称为“标准库”。每个模块都是一个Python程序，包含一组相关的函数，可以嵌入你的程序之中。例如，math模块有数学运算相关的函数，random模块有随机数相关的函数，等等。

在开始使用一个模块中的函数之前，必须用import语句导入该模块。在代码中，import语句包含以下部分：

- import关键字；
- 模块的名称；
- 可选的更多模块名称，之间用逗号隔开。

在导入一个模块后，就可以使用该模块中所有很酷的函数。让我们试一试random模块，它让我们能使用random.ranint()函数。

在文件编辑器中输入以下代码，保存为printRandom.py：

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

如果运行这个程序，输出看起来可能像这样：

```
4
1
```

```
8
4
1
```

random.randint()函数调用求值为传递给它的两个整数之间的一个随机整数。因为randint()属于random模块，必须在函数名称之前先加上random.，告诉python在random模块中寻找这个函数。

下面是import语句的例子，它导入了4个不同的模块：

```
import random, sys, os, math
```

现在我们可以使用这4个模块中的所有函数。本书后面我们将学习更多的相关内容。

from import语句

import语句的另一种形式包括from关键字，之后是模块名称，import关键字和一个星号，例如from random import *。

使用这种形式的import语句，调用random模块中的函数时不需要random.前缀。但是，使用完整的名称会让代码更可读，所以最好是使用普通形式的import语句。

2.9 用**sys.exit()**提前结束程序

要介绍的最后一个控制流概念，是如何终止程序。当程序执行到指令的底部时，总是会终止。但是，通过调用sys.exit()函数，可以让程序终止或退出。因为这个函数在sys模块中，所以必须先导入sys，才能使用它。

打开一个新的文件编辑器窗口，输入以下代码。保存为exitExample.py：

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

在IDLE中运行这个程序。该程序有一个无限循环，里面没有break语句。结束该程序的唯一方式，就是用户输入exit，导致sys.exit()被调用。如果response等于exit，程序就会中止。因为response变量由input()函数赋值，所以用户必须输入exit，才能停止该程序。

2.10 小结

通过使用求值为True或False的表达式（也称为条件），你可以编写程序来决定哪些代码执行，哪些代码跳过。可以在循环中一遍又一遍地执行代码，只要某个条件求值为True。如果需要跳出循环或回到开始处，break和continue语句很有用。

这些控制流语句让你写出非常聪明的程序。还有另一种类型的控制流，你可以通过编写自己的函数来实现。这是下一章的主题。

2.11 习题

1．布尔数据类型的两个值是什么？如何拼写？

2．3个布尔操作符是什么？

3．写出每个布尔操作符的真值表（也就是操作数的每种可能组合，以及操作的结果）。

4．以下表达式求值的结果是什么？

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5．6个比较操作符是什么？

6．等于操作符和赋值操作符的区别是什么？

7．解释什么是条件，可以在哪里使用条件。

8．识别这段代码中的3个语句块：

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9．编写代码，如果变量spam中存放1，就打印Hello，如果变量中存放2，就打印Howdy，如果变量中存放其他值，就打印Greetings!

10．如果程序陷在一个无限循环中，你可以按什么键？

11．break和continue之间的区别是什么？

12．在for循环中，range(10)、range(0, 10)和range(0, 10, 1)之间的区别是什么？

13．编写一小段程序，利用for循环，打印出从1到10的数字。然后利用while循环，编写一个等价的程序，打印出从1到10的数字。

14．如果在名为spam的模块中，有一个名为bacon()的函数，那么在导入spam模块后，如何调用它？

附加题：在因特网上查找round()和abs()函数，弄清楚它们的作用。在交互式环境中尝试使用它们。

第3章　函数

从前面的章节中，你已经熟悉了print()、input()和len()函数。Python提供了这样一些内建函数，但你也可以编写自己的函数。“函数”就像一个程序内的小程序。

为了更好地理解函数的工作原理，让我们来创建一个函数。在文件编辑器中输入下面的程序，保存为helloFunc.py：

```
❶ def hello():
❷     print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')

❸ hello()
  hello()
  hello()
```

第一行是def语句❶，它定义了一个名为hello()的函数。def语句之后的代码块是函数体❷。这段代码在函数调用时执行，而不是在函数第一次定义时执行。

函数之后的hello()语句行是函数调用❸。在代码中，函数调用就是函数名后跟上括号，也许在括号之间有一些参数。如果程序执行遇到这些调用，就会跳到函数的第一行，开始执行那里的代码。如果执行到达函数的末尾，就回到调用函数的那行，继续像以前一样向下执行代码。

因为这个程序调用了3次hello()函数，所以函数中的代码就执行了3次。在运行这个程序时，输出看起来像这样：

```
Howdy!
Howdy!!!
Hello there.
Howdy!
```

```
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

函数的一个主要目的就是将需要多次执行的代码放在一起。如果没有函数定义，你可能每次都需要复制粘贴这些代码，程序看起来可能会像这样：

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

一般来说，我们总是希望避免复制代码，因为如果一旦决定要更新代码（比如说，发现了一个缺陷要修复），就必须记住要修改所有复制的代码。

随着你获得更多的编程经验，常常会发现自己在为代码“消除重复”，即去除一些重复或复制的代码。消除重复能够使程序更短、更易读、更容易更新。

3.1 def语句和参数

如果调用print()或len()函数，你会传入一些值，放在括号之间，在这里称为“参数”。也可以自己定义接收参数的函数。在文件编辑器中输入这个例子，将它保存为helloFunc2.py：

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

如果运行这个程序，输出看起来像这样：

```
Hello Alice
Hello Bob
```

在这个程序的hello()函数定义中，有一个名为name的变元❶。“变元”是一个变量，当函数被调用时，参数就存放在其中。hello()函数第一次被调用时，使用的参数是'Alice'❸。程序执行进入该函数，变量name自动设为'Alice'，就是被print()语句打印出的内容❷。

关于变元有一件特殊的事情值得注意：保存在变元中的值，在函数返回后就丢失了。例如前面的程序，如果你在hello('Bob')之后添加print(name)，程序会报NameError，因为没有名为name的变量。在函数调用hello('Bob')返回后，这个变量被销毁了，所以print(name)会引用一个不存在的变量name。

这类似于程序结束时，程序中的变量会丢弃。在本章稍后，当我们探讨函数的局部作用域时，我会进一步分析为什么会这样。

3.2 返回值和return语句

如果调用len()函数，并向它传入像'Hello'这样的参数，函数调用就求值为整数5。这是传入的字符串的长度。一般来说，函数调用求值的结果，称为函数的“返回值”。

用def语句创建函数时，可以用return语句指定应该返回什么值。

return语句包含以下部分：

- return关键字；
- 函数应该返回的值或表达式。

如果在return语句中使用了表达式，返回值就是该表达式求值的结果。例如，下面的程序定义了一个函数，它根据传入的数字参数，返回一个不同的字符串。在文件编辑器中输入以下代码，并保存为magic8Ball.py：

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
           return 'It is certain'
       elif answerNumber == 2:
           return 'It is decidedly so'
       elif answerNumber == 3:
           return 'Yes'
       elif answerNumber == 4:
           return 'Reply hazy try again'
       elif answerNumber == 5:
           return 'Ask again later'
       elif answerNumber == 6:
           return 'Concentrate and ask again'
       elif answerNumber == 7:
           return 'My reply is no'
       elif answerNumber == 8:
           return 'Outlook not so good'
       elif answerNumber == 9:
           return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

在这个程序开始时，Python首先导入random模块❶。然后getAnswer()函数被定义❷。因为函数是被定义（而不是被调用），所以执行会跳过其中的代码。接下来，random.randint()函数被调用，带两个

参数，1和9❹。它求值为1和9之间的一个随机整数（包括1和9），这个值被存在一个名为r的变量中。

getAnswer()函数被调用，以r作为参数❺。程序执行转移到getAnswer()函数的顶部❸，r的值被保存到名为answerNumber的变元中。然后，根据answerNumber中的值，函数返回许多可能字符串中的一个。程序执行返回到程序底部的代码行，即原来调用getAnswer()的地方❺。返回的字符串被赋给一个名为fortune变量，然后它又被传递给print()调用❻，并被打印在屏幕上。

请注意，因为可以将返回值作为参数传递给另一个函数调用，所以你可以将下面3行代码

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

缩写成一行等价的代码：

```
print(getAnswer(random.randint(1, 9)))
```

记住，表达式是值和操作符的组合。函数调用可以用在表达式中，因为它求值为它的返回值。

3.3 None值

在Python中有一个值称为None，它表示没有值。None是NoneType数据类型的唯一值（其他编程语言可能称这个值为null、nil或undefined）。就像布尔值True和False一样，None必须大写首字母N。

如果你希望变量中存储的东西不会与一个真正的值混淆，这个没有

值的值就可能有用。有一个使用None的地方就是print()的返回值。print()函数在屏幕上显示文本，但它不需要返回任何值，这和len()或input()不同。但既然所有函数调用都需要求值为一个返回值，那么print()就返回None。要看到这个效果，请在交互式环境中输入以下代码。

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

在幕后，对于所有没有return语句的函数定义，Python都会在末尾加上return None。这类似于while或for循环隐式地以continue语句结尾。而且，如果使用不带值的return语句（也就是只有return关键字本身），那么就返回None。

3.4 关键字参数和print()

大多数参数是由它们在函数调用中的位置来识别的。例如，random.randint(1, 10)与random.randint(10, 1)不同。函数调用random.randint(1, 10)将返回1到10之间的一个随机整数，因为第一个参数是范围的下界，第二个参数是范围的上界（而random.randint(10, 1)会导致错误）。

但是，“关键字参数”是由函数调用时加在它们前面的关键字来识别的。关键字参数通常用于可选变元。例如，print()函数有可选的变元end

和sep，分别指定在参数末尾打印什么，以及在参数之间打印什么来隔开它们。

如果运行以下程序：

```
print('Hello')
print('World')
```

输出将会是：

```
Hello
World
```

这两个字符串出现在独立的两行中，因为print()函数自动在传入的字符串末尾添加了换行符。但是，可以设置end关键字参数，将它变成另一个字符串。例如，如果程序像这样：

```
print('Hello', end='')
print('World')
```

输出就会像这样：

```
HelloWorld
```

输出被打印在一行中，因为在'Hello'后面不再打印换行，而是打印

了一个空字符串。如果需要禁用加到每一个print()函数调用末尾的换行，这就很有用。

类似地，如果向print()传入多个字符串值，该函数就会自动用一个空格分隔它们。在交互式环境中输入以下代码：

```
>>> print('cats', 'dogs', 'mice')

cats dogs mice
```

但是你可以传入sep关键字参数，替换掉默认的分隔字符串。在交互式环境中输入以下代码：

```
>>> print('cats', 'dogs', 'mice', sep=',')

cats,dogs,mice
```

也可以在你编写的函数中添加关键字参数，但必须先在接下来的两章中学习列表和字典数据类型。现在只要知道，某些函数有可选的关键字参数，在函数调用时可以指定。

3.5 局部和全局作用域

在被调用函数内赋值的变元和变量，处于该函数的“局部作用域”。在所有函数之外赋值的变量，属于“全局作用域”。处于局部作用域的变量，被称为“局部变量”。处于全局作用域的变量，被称为“全局变量”。一个变量必是其中一种，不能既是局部的又是全局的。

可以将“作用域”看成是变量的容器。当作用域被销毁时，所有保存在该作用域内的变量的值就被丢弃了。只有一个全局作用域，它是在程序开始时创建的。如果程序终止，全局作用域就被销毁，它的所有变量就被丢弃了。否则，下次你运行程序的时候，这些变量就会记住它们上次运行时的值。

一个函数被调用时，就创建了一个局部作用域。在这个函数内赋值的所有变量，存在于该局部作用域内。该函数返回时，这个局部作用域就被销毁了，这些变量就丢失了。下次调用这个函数，局部变量不会记得该函数上次被调用时它们保存的值。

作用域很重要，理由如下：

- 全局作用域中的代码不能使用任何局部变量；
- 但是，局部作用域可以访问全局变量；
- 一个函数的局部作用域中的代码，不能使用其他局部作用域中的变量。
- 如果在不同的作用域中，你可以用相同的名字命名不同的变量。也就是说，可以有一个名为spam的局部变量，和一个名为spam的全局变量。

Python有不同的作用域，而不是让所有东西都成全局变量，这是有理由的。这样一来，当特定函数调用中的代码修改变量时，该函数与程序其他部分的交互，只能通过它的参数和返回值。这缩小了可能导致缺陷的代码作用域。如果程序只包含全局变量，又有一个变量赋值错误的缺陷，那就很难追踪这个赋值错误发生的位置。它可能在程序的任何地方赋值，而你的程序可能有几百到几千行！但如果缺陷是因为局部变量错误赋值，你就会知道，只有那一个函数中的代码可能产生赋值错误。

虽然在小程序中使用全局变量没有太大问题，但当程序变得越来越大时，依赖全局变量就是一个坏习惯。

3.5.1 局部变量不能在全局作用域内使用

考虑下面的程序，它在运行时会产生错误：

```
def spam():
    eggs = 31337
spam()
print(eggs)
```

如果运行这个程序，输出将是：

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

发生错误是因为，eggs变量只属于spam()调用所创建的局部作用域。在程序执行从spam返回后，该局部作用域就被销毁了，不再有名为eggs的变量。所以当程序试图执行print(eggs)，Python就报错，说eggs没有定义。你想想看，这是有意义的。当程序执行在全局作用域中时，不存在局部作用域，所以不会有任何局部变量。这就是为什么只有全局变量能用于全局作用域。

3.5.2 局部作用域不能使用其他局部作用域内的变量

一个函数被调用时，就创建了一个新的局部作用域，这包括一个函数被另一个函数调用时的情况。请看以下代码：

```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0
```

```
❺ spam()
```

在程序开始运行时，spam()函数被调用❺，创建了一个局部作用域。局部变量eggs❶被赋值为99。然后bacon()函数被调用❷，创建了第二个局部作用域。多个局部作用域能同时存在。在这个新的局部作用域中，局部变量ham被赋值为101。局部变量eggs（与spam()的局部作用域中的那个变量不同）也被创建❹，并赋值为0。

当bacon()返回时，这次调用的局部作用域被销毁。程序执行在spam()函数中继续，打印出eggs的值❸。因为spam()调用的局部作用域仍然存在，eggs变量被赋值为99。这就是程序的打印输出。

要点在于，一个函数中的局部变量完全与其他函数中的局部变量分隔开来。

3.5.3 全局变量可以在局部作用域中读取

请看以下程序：

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

因为在spam()函数中，没有变元名为eggs，也没有代码为eggs赋值，所以当spam()中使用eggs时，Python认为它是对全局变量eggs的引用。这就是前面的程序运行时打印出42的原因。

3.5.4 名称相同的局部变量和全局变量

要想生活简单，就要避免局部变量与全局变量或其他局部变量同名。但在技术上，在Python中让局部变量和全局变量同名是完全合法的。为了看看实际发生的情况，请在文件编辑器中输入以下代码，并保存为sameName.py：

```
  def spam():
❶     eggs = 'spam local'
      print(eggs) # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs) # prints 'bacon local'
      spam()
      print(eggs) # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs) # prints 'global'
```

运行该程序，输出如下：

```
bacon local
spam local
bacon local
global
```

在这个程序中，实际上有3个不同的变量，但令人迷惑的是，它们都名为eggs。这些变量是：

❶名为eggs的变量，存在于spam()被调用时的局部作用域；

❷名为eggs的变量，存在于bacon()被调用时的局部作用域；

❸名为eggs的变量，存在于全局作用域。

因为这3个独立的变量都有相同的名字，追踪某一个时刻使用的是哪个变量，可能比较麻烦。这就是应该避免在不同作用域内使用相同变量名的原因。

3.6 global语句

如果需要在一个函数内修改全局变量，就使用global语句。如果在函数的顶部有global eggs这样的代码，它就告诉Python，“在这个函数中，eggs指的是全局变量，所以不要用这个名字创建一个局部变量。”例如，在文件编辑器中输入以下代码，并保存为sameName2.py：

```
  def spam():
❶     global eggs
❷     eggs = 'spam'

  eggs = 'global'
  spam()
  print(eggs)
```

运行该程序，最后的print()调用将输出：

```
spam
```

因为eggs在spam()的顶部被声明为global❶，所以当eggs被赋值为'spam'时❷，赋值发生在全局作用域的spam上。没有创建局部spam变量。

有4条法则，来区分一个变量是处于局部作用域还是全局作用域：

1．如果变量在全局作用域中使用（即在所有函数之外），它就总是全局变量。

2．如果在一个函数中，有针对该变量的global语句，它就是全局变量。

3．否则，如果该变量用于函数中的赋值语句，它就是局部变量。

4．但是，如果该变量没有用在赋值语句中，它就是全局变量。

为了更好地理解这些法则，这里有一个例子程序。在文件编辑器中输入以下代码，并保存为sameName3.py：

```
   def spam():
❶    global eggs
     eggs = 'spam' # this is the global

   def bacon():
❷    eggs = 'bacon' # this is a local
   def ham():
❸    print(eggs) # this is the global

   eggs = 42 # this is the global
   spam()
   print(eggs)
```

在spam()函数中，eggs是全局eggs变量，因为在函数的开始处，有针对eggs变量的global语句❶。在bacon()中，eggs是局部变量，因为在该函数中有针对它的赋值语句❷。在ham()中❸，eggs是全局变量，因为在这个函数中，既没有赋值语句，也没有针对它的global语句。如果运行sameName3.py，输出将是：

```
spam
```

在一个函数中，一个变量要么总是全局变量，要么总是局部变量。函数中的代码没有办法先使用名为eggs的局部变量，稍后又在同一个函数中使用全局eggs变量。

如果想在一个函数中修改全局变量中存储的值，就必须对该变量使用global语句。

在一个函数中，如果试图在局部变量赋值之前就使用它，像下面的程序这样，Python就会报错。为了看到效果，请在文件编辑器中输入以下代码，并保存为sameName4.py：

```
  def spam():
      print(eggs) # ERROR!
❶     eggs = 'spam local'

❷ eggs = 'global'
  spam()
```

运行前面的程序，会产生出错信息。

```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

发生这个错误是因为，Python看到spam()函数中有针对eggs的赋值语句❶，因此认为eggs变量是局部变量。但是因为print(eggs)的执行在eggs赋值之前，局部变量eggs并不存在。Python不会退回到使用全局eggs变量❷。

3.7 异常处理

到目前为止，在Python程序中遇到错误，或“异常”，意味着整个程序崩溃。你不希望这发生在真实世界的程序中。相反，你希望程序能检测错误，处理它们，然后继续运行。

例如，考虑下面的程序，它有一个“除数为零”的错误。打开一个新的文件编辑器窗口，输入以下代码，并保存为zeroDivide.py：

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

我们已经定义了名为spam的函数，给了它一个变元，然后打印出该函数带各种参数的值，看看会发生什么情况。下面是运行前面代码的输出：

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

当试图用一个数除以零时，就会发生ZeroDivisionError。根据错误信息中给出的行号，我们知道spam()中的return语句导致了一个错误。

函数作为“黑盒”

通常，对于一个函数，你要知道的就是它的输入值（变元）和输出值。你并非总是需要加重自己的负担，弄清楚函数的代码实际是怎样工作的。如果以这种高层的方式来思考函数，通常大家会说，你将该函数看成是一个黑盒。

这个思想是现代编程的基础。本书后面的章节将向你展示一些模块，其中的函数是由其他人编写的。尽管你在好奇的时候也可以看一看源代码，但为了能使用它们，你并不需要知道它们是如何工作的。而且，因为鼓励在编写函数时不使用全局变量，你通常也不必担心函数的代码会与程序的其他部分发生交叉影响。

错误可以由try和except语句来处理。那些可能出错的语句被放在try子句中。如果错误发生，程序执行就转到接下来的except子句开始处。

可以将前面除数为零的代码放在一个try子句中，让except子句包含代码，来处理该错误发生时应该做的事。

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

如果在try子句中的代码导致一个错误，程序执行就立即转到except子句的代码。在运行那些代码之后，执行照常继续。前面程序的输出如下：

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

请注意，在函数调用中的try语句块中，发生的所有错误都会被捕捉。请考虑以下程序，它的做法不一样，将spam()调用放在语句块中：

```
def spam(divideBy):
    return 42 / divideBy

try:
```

```
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

该程序运行时，输出如下：

```
21.0
3.5
Error: Invalid argument.
```

print(spam(1))从未被执行是因为，一旦执行跳到except子句的代码，就不会回到try子句。它会继续照常向下执行。

3.8 一个小程序：猜数字

到目前为止，前面展示的小例子适合于介绍基本概念。现在让我们看一看，如何将所学的知识综合起来，编写一个更完整的程序。在本节中，我将展示一个简单的猜数字游戏。在运行这个程序时，输出看起来像这样：

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
```

```
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

在文件编辑器中输入以下代码，并保存为guessTheNumber.py：

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break # This condition is the correct guess!
```

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' gue
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

让我们逐行来看看代码，从头开始。

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

首先，代码顶部的一行注释解释了这个程序做什么。然后，程序导入了模块random，以便能用random.randint()函数生成一个数字，让用户来猜。返回值是一个1到20之间的随机整数，保存在变量secretNumber中。

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

程序告诉玩家，它有了一个秘密数字，并且给玩家6次猜测机会。在for循环中，代码让玩家输入一次猜测，并检查该猜测。该循环最多迭代6次。循环中发生的第一件事情，是让玩家输入一个猜测数字。因为input()返回一个字符串，所以它的返回值被直接传递给int()，它将字符串转变成整数。这保存在名为guess的变量中。

```
    if guess < secretNumber:
```

```
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
```

这几行代码检查该猜测是小于还是大于那个秘密数字。不论哪种情况，都在屏幕上打印提示。

```
    else:
        break # This condition is the correct guess!
```

如果该猜测既不大于也不小于秘密数字，那么它就一定等于秘密数字，这时你希望程序执行跳出for循环。

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' gue
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

在for循环后，前面的if...else语句检查玩家是否正确地猜到了该数字，并将相应的信息打印在屏幕上。不论哪种情况，程序都会打印一个包含整数值的变量（guessesTaken和secretNumber）。因为必须将这些整数值连接成字符串，所以它将这些变量传递给str()函数，该函数返回这些整数值的字符串形式。现在这些字符串可以用+操作符连接起来，最后传递给print()函数调用。

3.9 小结

函数是将代码逻辑分组的主要方式。因为函数中的变量存在于它们自己的局部作用域内，所以一个函数中的代码不能直接影响其他函数中

变量的值。这限制了哪些代码才能改变变量的值，对于调试代码是很有帮助的。

函数是很好的工具，帮助你组织代码。你可以认为他们是黑盒。它们以参数的形式接收输入，以返回值的形式产生输出。它们内部的代码不会影响其他函数中的变量。

在前面几章中，一个错误就可能导致程序崩溃。在本章中，你学习了try和except语句，它们在检测到错误时会运行代码。这让程序在面对常见错误时更有灵活性。

3.10 习题

1．为什么在程序中加入函数会有好处？

2．函数中的代码何时执行：在函数被定义时，还是在函数被调用时？

3．什么语句创建一个函数？

4．一个函数和一次函数调用有什么区别？

5．Python程序中有多少全局作用域？有多少局部作用域？

6．当函数调用返回时，局部作用域中的变量发生了什么？

7．什么是返回值？返回值可以作为表达式的一部分吗？

8．如果函数没有返回语句，对它调用的返回值是什么？

9．如何强制函数中的一个变量指的是全局变量？

10．None的数据类型是什么？

11．import areallyourpetsnamederic语句做了什么？

12．如果在名为spam的模块中，有一个名为bacon()的函数，在引入spam后，如何调用它？

13．如何防止程序在遇到错误时崩溃？

14．try子句中发生了什么？except子句中发生了什么？

3.11 实践项目

作为实践，请编写程序完成下列任务。

3.11.1 Collatz序列

编写一个名为collatz()的函数，它有一个名为number的参数。如果参数是偶数，那么collatz()就打印出number // 2，并返回该值。如果number是奇数，collatz()就打印并返回3 * number + 1。

然后编写一个程序，让用户输入一个整数，并不断对这个数调用collatz()，直到函数返回值 1（令人惊奇的是，这个序列对于任何整数都有效，利用这个序列，你迟早会得到1！既使数学家也不能确定为什么。你的程序在研究所谓的“Collatz序列”，它有时候被称为“最简单的、不可能的数学问题”）。

记得将input()的返回值用int()函数转成一个整数，否则它会是一个字符串。

> 提示
>
> 如果number % 2 == 0，整数number就是偶数，如果number % 2 == 1，它就是奇数。

这个程序的输出看起来应该像这样：

```
Enter number:
3
10
5
16
8
4
2
1
```

3.11.2 输入验证

在前面的项目中添加try和except语句，检测用户是否输入了一个非整数的字符串。正常情况下，int()函数在传入一个非整数字符串时，会产生ValueError错误，比如int('puppy')。在except子句中，向用户输出一条信息，告诉他们必须输入一个整数。

第4章　列表

在你能够开始编写程序之前，还有一个主题需要理解，那就是列表数据类型及元组。列表和元组可以包含多个值，这样编写程序来处理大量数据就变得更容易。而且，由于列表本身又可以包含其他列表，所以可以用它们将数据安排成层次结构。

本章将探讨列表的基础知识。我也会讲授关于方法的内容。方法也是函数，它们与特定数据类型的值绑定。然后我会简单介绍类似列表的元组和字符串数据类型，以及它们与列表值的比较。下一章将介绍字典数据类型。

4.1 列表数据类型

"列表"是一个值，它包含多个字构成的序列。术语"列表值"指的是列表本身（它作为一个值，可以保存在变量中，或传递给函数，像所有其他值一样），而不是指列表值之内的那些值。列表值看起来像这样：['cat', 'bat', 'rat', 'elephant']。就像字符串值用引号来标记字符串的起止一样，列表用左方括号开始，右方括号结束，即[]。列表中的值也称为"表项"。表项用逗号分隔（就是说，它们是"逗号分隔的"）。例如，在交互式环境中输入以下代码：

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
```

```
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

spam变量❶仍然只被赋予一个值：列表值。但列表值本身包含多个值。[]是一个空列表，不包含任何值，类似于空字符串''。

4.1.1 用下标取得列表中的单个值

假定列表['cat', 'bat', 'rat', 'elephant']保存在名为spam的变量中。Python代码spam[0]将求值为'cat'，spam[1]将求值为'bat'，依此类推。列表后面方括号内的整数被称为"下标"。列表中第一个值的下标是0，第二个值的下标是1，第三个值的下标是2，依此类推。图4-1展示了一个赋给spam的列表值，以及下标表达式的求值结果。

图4-1 一个列表值保存在spam变量中，展示了每个下标指向哪个值

例如，在交互式环境中输入以下表达式。开始将列表赋给变量spam。

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
```

```
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

请注意，表达式'Hello ' + spam[0] ❶求值为'Hello ' + 'cat'，因为spam[0]求值为字符串'cat'。这个表达式也因此求值为字符串'Hello cat'❷。

如果使用的下标超出了列表中值的个数，Python将给出IndexError出错信息。

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

下标只能是整数，不能是浮点值。下面的例子将导致TypeError错误：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

列表也可以包含其他列表值。这些列表的列表中的值，可以通过多重下标来访问，像这样：

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
```

```
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

第一个下标表明使用哪个列表值，第二个下标表明该列表值中的值。例如，spam[0][1]打印出'bat'，即第一个列表中的第二个值。如果只使用一个下标，程序将打印出该下标处的完整列表值。

4.1.2 负数下标

虽然下标从0开始并向上增长，但也可以用负整数作为下标。整数值−1指的是列表中的最后一个下标，−2指的是列表中倒数第二个下标，以此类推。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

4.1.3 利用切片取得子列表

就像下标可以从列表中取得单个值一样，“切片”可以从列表中取得多个值，结果是一个新列表。切片输入在一对方括号中，像下标一样，但它有两个冒号分隔的整数。请注意下标和切片的不同。

- spam[2]是一个列表和下标（一个整数）。
- spam[1:4]是一个列表和切片（两个整数）。

在一个切片中，第一个整数是切片开始处的下标。第二个整数是切片结束处的下标。切片向上增长，直至第二个下标的值，但不包括它。切片求值为一个新的列表值。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

作为快捷方法，你可以省略切片中冒号两边的一个下标或两个下标。省略第一个下标相当于使用0，或列表的开始。省略第二个下标相当于使用列表的长度，意味着分片直至列表的末尾。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
```

```
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

4.1.4 用**len()**取得列表的长度

len()函数将返回传递给它的列表中值的个数，就像它能计算字符串中字符的个数一样。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

4.1.5 用下标改变列表中的值

一般情况下，赋值语句左边是一个变量名，就像spam = 4。但是，也可以使用列表的下标来改变下标处的值。例如，spam[1] = 'aardvark'意味着“将列表spam下标1处的值赋值为字符串'aardvark'。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
```

```
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

4.1.6 列表连接和列表复制

+操作符可以连接两个列表，得到一个新列表，就像它将两个字符串合并成一个新字符串一样。*操作符可以用于一个列表和一个整数，实现列表的复制。在交互式环境中输入以下代码：

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2,  3,  'A',  'B',  'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
```

```
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

4.1.7 用**del**语句从列表中删除值

del语句将删除列表中下标处的值，表中被删除值后面的所有值，都将向前移动一个下标。例如，在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
```

```
['cat', 'bat']
```

del语句也可用于一个简单变量，删除它，作用就像是“取消赋值”语句。如果在删除之后试图使用该变量，就会遇到NameError错误，因为该变量已不再存在。

在实践中，你几乎永远不需要删除简单变量。del语句几乎总是用于删除列表中的值。

4.2 使用列表

当你第一次开始编程时，很容易会创建许多独立的变量，来保存一组类似的值。例如，如果要保存我的猫的名字，可能会写出这样的代码：

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

事实表明这是一种不好的编程方式。举一个例子，如果猫的数目发生改变，程序就不得不增加变量，来保存更多的猫。这种类型的程序也有很多重复或几乎相等的代码。考虑下面的程序中有多少重复代码，在文本编辑器中输入它并保存为allMyCats1.py：

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
```

```
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

不必使用多个重复的变量，你可以使用单个变量，包含一个列表值。例如，下面是新的改进版本的allMyCats1.py程序。这个新版本使用了一个列表，可以保存用户输入的任意多的猫。在新的文件编辑器窗口中，输入以下代码并保存为allMyCats2.py。

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name] # list concatenation
print('The cat names are:')
for name in catNames:
    print(' ' + name)
```

运行这个程序，输出看起来像这样：

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
```

```
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
```

```
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
```

```
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
```

```
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
```

```
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
```

```
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
  Fat-tail
  Miss Cleo
```

使用列表的好处在于，现在数据放在一个结构中，所以程序能够更灵活的处理数据，比放在一些重复的变量中方便。

4.2.1 列表用于循环

在第2章中，你学习了使用循环，对一段代码执行一定次数。从技术上说，循环是针对一个列表或类似列表中的每个值，重复地执行代码块。例如，如果执行以下代码：

```
for i in range(4):
    print(i)
```

程序的输出将是：

```
0
1
2
3
```

这是因为range(4)的返回值是类似列表的值。Python认为它类似于[0, 1, 2, 3]。下面的程序和前面的程序输出相同：

```
for i in [0, 1, 2, 3]:
    print(i)
```

前面的for循环实际上是在循环执行它的子句，在每次迭代中，让变量依次设置为列表中的值。

注意

在本书中，我使用术语“类似列表”，来指技术上称为“序列”的数据类型。但是，你不需要知道这个术语的技术定义。

一个常见的Python技巧，是在for循环中使用range(len(someList))，迭代列表的每一个下标。例如，在交互式环境中输入以下代码：

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

在前面的循环中使用range(len(supplies))很方便，这是因为，循环中的代码可以访问下标（通过变量i），以及下标处的值（通过supplies[i]）。最妙的是，range(len(supplies))将迭代supplies的所有下标，无论它包含多少表项。

4.2.2 in和not in操作符

利用in和not in操作符，可以确定一个值否在列表中。像其他操作符一样，in和not in用在表达式中，连接两个值：一个要在列表中查找的值，以及待查找的列表。这些表达式将求值为布尔值。在交互式环境中输入以下代码：

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
```

```
True
```

例如，下面的程序让用户输入一个宠物名字，然后检查该名字是否在宠物列表中。打开一个新的文件编辑器窗口，输入以下代码，并保存为myPets.py：

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

输出可能像这样：

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

4.2.3 多重赋值技巧

多重赋值技巧是一种快捷方式，让你在一行代码中，用列表中的值

为多个变量赋值。所以不必像这样：

```
>>> cat = ['fat', 'black', 'loud']

>>> size = cat[0]

>>> color = cat[1]

>>> disposition = cat[2]
```

而是输入下面的代码：

```
>>> cat = ['fat', 'black', 'loud']

>>> size, color, disposition = cat
```

变量的数目和列表的长度必须严格相等，否则Python将给出ValueError：

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

4.3 增强的赋值操作

在对变量赋值时，常常会用到变量本身。例如，将42赋给变量spam之后，用下面的代码让spam的值增加1：

```
>>> spam = 42
>>> spam = spam + 1
```

```
>>> spam
43
```

作为一种快捷方式，可以用增强的赋值操作符+=来完成同样的事：

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

针对+、-、*、/和%操作符，都有增强的赋值操作符，如表4-1所

示。

表**4-1** 增强的赋值操作符

增强的赋值语句	等价的赋值语句
spam += 1	spam = spam + 1
spam -= 1	spam = spam - 1
spam *= 1	spam = spam * 1
spam /= 1	spam = spam / 1
spam %= 1	spam = spam % 1

+=操作符也可以完成字符串和列表的连接，*=操作符可以完成字符串和列表的复制。在交互式环境中输入以下代码：

```
>>> spam = 'Hello'

>>> spam += ' world!'

>>> spam

'Hello world!'
```

```
>>> bacon = ['Zophie']

>>> bacon *= 3

>>> bacon

['Zophie', 'Zophie', 'Zophie']
```

4.4 方法

方法和函数是一回事，只是它是调用在一个值上。例如，如果一个列表值存储在spam中，你可以在这个列表上调用index()列表方法（稍后我会解释），就像spam.index('hello')一样。方法部分跟在这个值后面，以一个句点分隔。

每种数据类型都有它自己的一组方法。例如，列表数据类型有一些有用的方法，用来查找、添加、删除或操作列表中的值。

4.4.1 用**index()**方法在列表中查找值

列表值有一个index()方法，可以传入一个值，如果该值存在于列表中，就返回它的下标。如果该值不在列表中，Python就报ValueError。在交互式环境中输入以下代码：

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
```

```
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

如果列表中存在重复的值，就返回它第一次出现的下标。在交互式环境中输入以下代码，注意index()返回1，而不是3：

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
```

```
1
```

4.4.2 用**append()**和**insert()**方法在列表中添加值

要在列表中添加新值，就使用append()和 insert()方法。在交互式环境中输入以下代码，对变量spam中的列表调用append()方法：

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.append('moose')

>>> spam

['cat', 'dog', 'bat', 'moose']
```

前面的append()方法调用，将参数添加到列表末尾。insert()方法可以在列表任意下标处插入一个值。insert()方法的第一个参数是新值的下标，第二个参数是要插入的新值。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.insert(1, 'chicken')

>>> spam

['cat', 'chicken', 'dog', 'bat']
```

请注意，代码是spam.append('moose')和spam.insert(1, 'chicken')，而不是spam = spam.append('moose')和spam = spam.insert(1, 'chicken')。append()和insert()都不会将spam的新值作为其返回值（实际上，append()和insert()的返回值是None，所以你肯定不希望将它保存为变量的新值）。但是，列表被“当场”修改了。在4.6.1节“可变和不变数据类型”中，将更详细地介绍当场修改一个列表。

方法属于单个数据类型。append()和insert()方法是列表方法，只能在列表上调用，不能在其他值上调用，例如字符串和整型。在交互式环境中输入以下代码，注意产生的AttributeError错误信息：

```
>>> eggs = 'hello'

>>> eggs.append('world')
```

```
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

4.4.3 用**remove()**方法从列表中删除值

给 remove()方法传入一个值，它将从被调用的列表中删除。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
```

```
>>> spam
['cat', 'rat', 'elephant']
```

试图删除列表中不存在的值，将导致ValueError错误。例如，在交互式环境中输入以下代码，注意显示的错误：

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

如果该值在列表中出现多次，只有第一次出现的值会被删除。在交互式环境中输入以下代码：

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
```

```
>>> spam.remove('cat')

>>> spam

['bat', 'rat', 'cat', 'hat', 'cat']
```

如果知道想要删除的值在列表中的下标，del语句就很好用。如果知道想要从列表中删除的值，remove()方法就很好用。

4.4.4 用sort()方法将列表中的值排序

数值的列表或字符串的列表，能用sort()方法排序。例如，在交互式环境中输入以下代码：

```
>>> spam = [2, 5, 3.14, 1, -7]

>>> spam.sort()

>>> spam
```

```
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

也可以指定reverse关键字参数为True，让sort()按逆序排序。在交互式环境中输入以下代码：

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

关于sort()方法，你应该注意3件事。首先，sort()方法当场对列表排序。不要写出spam = spam.sort()这样的代码，试图记录返回值。

其次，不能对既有数字又有字符串值的列表排序，因为Python不知道如何比较它们。在交互式环境中输入以下代码，注意TypeError错误：

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

第三，sort()方法对字符串排序时，使用“ASCII字符顺序”，而不是实际的字典顺序。这意味着大写字母排在小写字母之前。因此在排序时，小写的a在大写的Z之后。例如，在交互式环境中输入以下代码：

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
```

```
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

如果需要按照普通的字典顺序来排序，就在sort()方法调用时，将关键字参数key设置为str.lower。

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

这将导致sort()方法将列表中所有的表项当成小写，但实际上并不会改变它们在列表中的值。

4.5 例子程序：神奇8球和列表

前一章我们写过神奇8球程序。利用列表，可以写出更优雅的版本。不是用一些几乎一样的elif语句，而是创建一个列表，针对它编码。打开一个新的文件编辑器窗口，输入以下代码，并保存为magic8Ball2.py：

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

Python中缩进规则的例外

在大多数情况下，代码行的缩进告诉Python它属于哪一个代码块。但是，这个规则有几个例外。例如在源代码文件中，列表实际上可以跨越几行。这些行的缩进并不重要。Python知道，没有看到结束方括号，列表就没有结束。例如，代码可以看起来像这样：

```
spam = ['apples',
    'oranges',
                    'bananas',
'cats']
print(spam)
```

当然，从实践的角度来说，大部分人会利用Python的行为，让他们的列表看起来漂亮且可读，就像神奇8球程序中的消息列表一样。

> 也可以在行末使用续行字符\，将一条指令写成多行。可以把\看成是“这条指令在下一行继续”。\续行字符之后的一行中，缩进并不重要。例如，下面是有效的Python代码：
>
> ```
> print('Four score and seven ' + \
>       'years ago...')
> ```
>
> 如果希望将一长行的Python代码安排得更为可读，这些技巧是有用的。

运行这个程序，你会看到它与前面的magic8Ball.py程序效果一样。

请注意用作messages下标的表达式：random.randint(0, len(messages) - 1)。这产生了一个随机数作为下标，不论messages的大小是多少。也就是说，你会得到0与len(messages) - 1之间的一个随机数。这种方法的好处在于，很容易向列表添加或删除字符串，而不必改变其他行的代码。如果稍后更新代码，就可以少改几行代码，引入缺陷的可能性也更小。

4.6 类似列表的类型：字符串和元组

列表并不是唯一表示序列值的数据类型。例如，字符串和列表实际上很相似，只要你认为字符串是单个文本字符的列表。对列表的许多操作，也可以作用于字符串：按下标取值、切片、用于for循环、用于len()，以及用于in和not in操作符。为了看到这种效果，在交互式环境中输入以下代码：

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
```

```
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')
```

```
* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

4.6.1 可变和不可变数据类型

但列表和字符串在一个重要的方面是不同的。列表是“可变的”数据类型，它的值可以添加、删除或改变。但是，字符串是“不可变的”，它不能被更改。尝试对字符串中的一个字符重新赋值，将导致TypeError错误。在交互式环境中输入以下代码，你就会看到：

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

“改变”一个字符串的正确方式，是使用切片和连接。构造一个“新

的”字符串，从老的字符串那里复制一些部分。在交互式环境中输入以下代码：

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'
```

我们用[0:7]和[8:12]来指那些不想替换的字符。请注意，原来的'Zophie a cat'字符串没有被修改，因为字符串是不可变的。尽管列表值是可变的，但下面代码中的第二行并没有修改列表eggs：

```
>>> eggs = [1, 2, 3]
```

```
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

这里eggs中的列表值并没有改变，而是整个新的不同的列表值([4, 5, 6])，覆写了老的列表值。如图4-2所示。

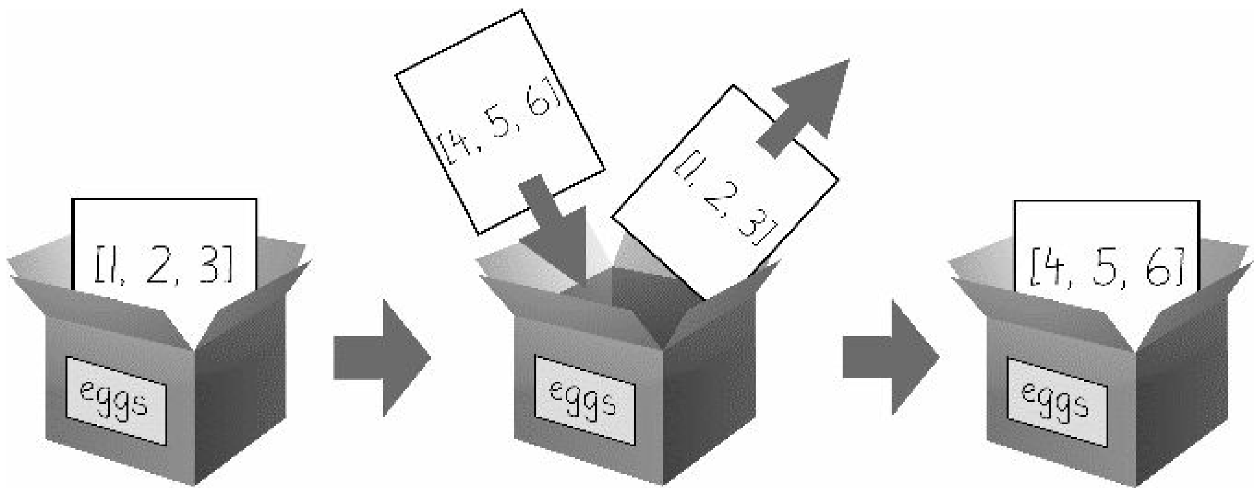


图4-2 当eggs = [4, 5, 6]被执行时，eggs的内容被新的列表值取代

如果你确实希望修改eggs中原来的列表，让它包含[4, 5, 6]，就要这样做：

```
>>> eggs = [1, 2, 3]
```

```
>>> del eggs[2]

>>> del eggs[1]

>>> del eggs[0]

>>> eggs.append(4)

>>> eggs.append(5)

>>> eggs.append(6)

>>> eggs

[4, 5, 6]
```

在第一个例子中，eggs最后的列表值与它开始的列表值是一样的。只是这个列表被改变了，而不是被覆写。图4-3展示了前面交互式脚本的例子中，前7行代码所做的7次改动。

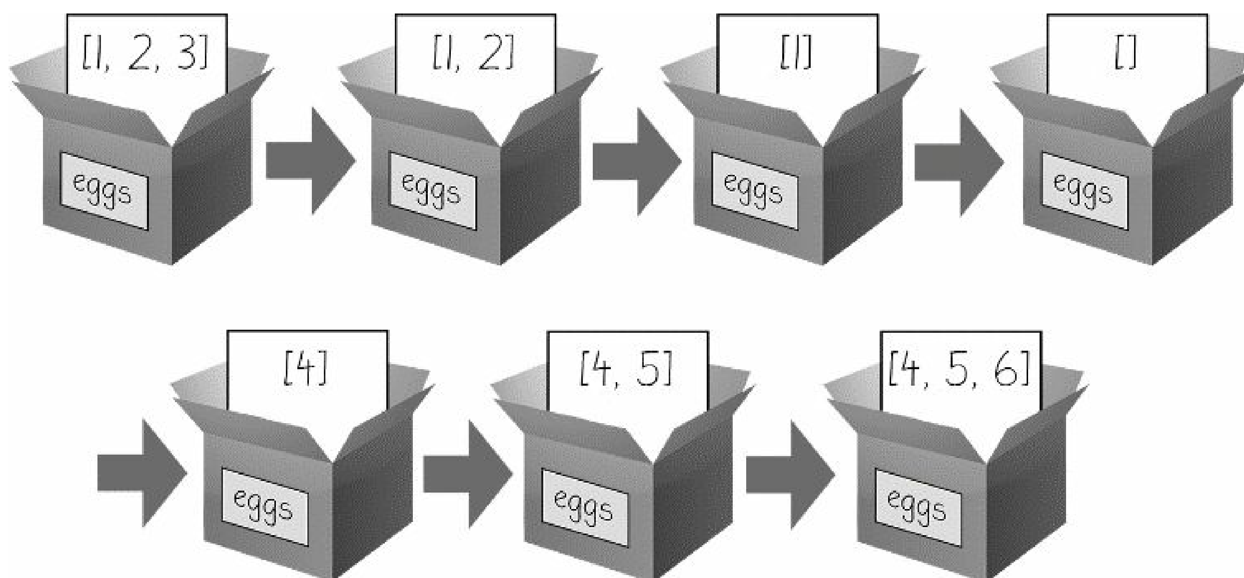


图4-3 del语句和append()方法当场修改了同一个列表值

改变一个可变数据类型的值（就像前面例子中del语句和append()方法所做的事），当场改变了该值，因为该变量的值没有被一个新的列表值取代。

区分可变与不可变类型，似乎没有什么意义，但4.7.1节“传递引用”将解释，使用可变参数和不可变参数调用函数时产生的不同行为。首先，让我们来看看元组数据类型，它是列表数据类型的不可变形式。

4.6.2 元组数据类型

除了两个方面，“元组”数据类型几乎与列表数据类型一样。首先，元组输入时用圆括号()，而不是用方括号[]。例如，在交互式环境中输入以下代码：

```
>>> eggs = ('hello', 42, 0.5)
```

```
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

但元组与列表的主要区别还在于，元组像字符串一样，是不可变的。元组不能让它们的值被修改、添加或删除。在交互式环境中输入以下代码，注意TypeError出错信息：

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
```

```
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

如果元组中只有一个值，你可以在括号内该值的后面跟上一个逗号，表明这种情况。否则，Python将认为，你只是在一个普通括号内输入了一个值。逗号告诉Python，这是一个元组（不像其他编程语言，Python接受列表或元组中最后表项后面跟的逗号）。在交互式环境中，输入以下的type()函数调用，看看它们的区别：

```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

你可以用元组告诉所有读代码的人，你不打算改变这个序列的值。如果需要一个永远不会改变的值的序列，就使用元组。使用元组而不是列表的第二个好处在于，因为它们是不可变的，它们的内容不会变化，Python可以实现一些优化，让使用元组的代码比使用列表的代码更快。

4.6.3 用list()和tuple()函数来转换类型

正如str(42)将返回'42'，即整数42的字符串表示形式，函数list()和tuple()将返回传递给它们的值的列表和元组版本。在交互式环境中输入以下代码，注意返回值与传入值是不同的数据类型：

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

如果需要元组值的一个可变版本，将元组转换成列表就很方便。

4.7 引用

正如你看到的，变量保存字符串和整数值。在交互式环境中输入以下代码：

```
>>> spam = 42
```

```
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

你将42赋给spam变量，然后拷贝spam中的值，将它赋给变量cheese。当稍后将spam中的值改变为100时，这不会影响cheese中的值。这是因为spam和cheese是不同的变量，保存了不同的值。

但列表不是这样的。当你将列表赋给一个变量时，实际上是将列表的“引用”赋给了该变量。引用是一个值，指向某些数据。列表引用是指向一个列表的值。这里有一些代码，让这个概念更容易理解。在交互式环境中输入以下代码：

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

这可能让你感到奇怪。代码只改变了cheese列表，但似乎cheese和spam列表同时发生了改变。

当创建列表时❶，你将对它的引用赋给了变量。但下一行❷只是将spam中的列表引用拷贝到cheese，而不是列表值本身。这意味着存储在spam和cheese中的值，现在指向了同一个列表。底下只有一个列表，因为列表本身实际从未复制。所以当你修改cheese变量的第一个元素时

❸，也修改了spam指向的同一个列表。

记住，变量就像包含着值的盒子。本章前面的图显示列表在盒子中，这并不准确，因为列表变量实际上没有包含列表，而是包含了对列表的“引用”（这些引用包含一些ID数字，Python在内部使用这些ID，但是你可以忽略）。利用盒子作为变量的隐喻，图4-4展示了列表被赋给spam变量时发生的情形。

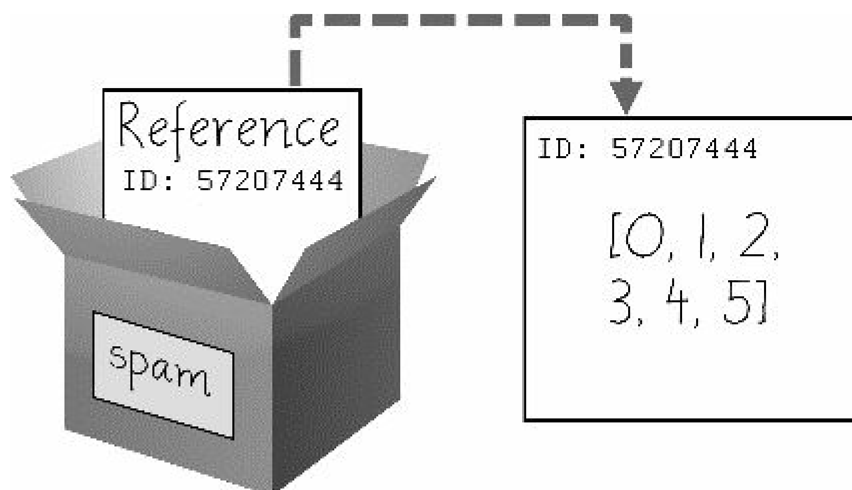


图4-4 spam = [0, 1, 2, 3, 4, 5]保存了对列表的引用，而非实际列表

然后，在图4-5中，spam中的引用被复制给cheese。只有新的引用被创建并保存在cheese中，而非新的列表。请注意，两个引用都指向同一个列表。

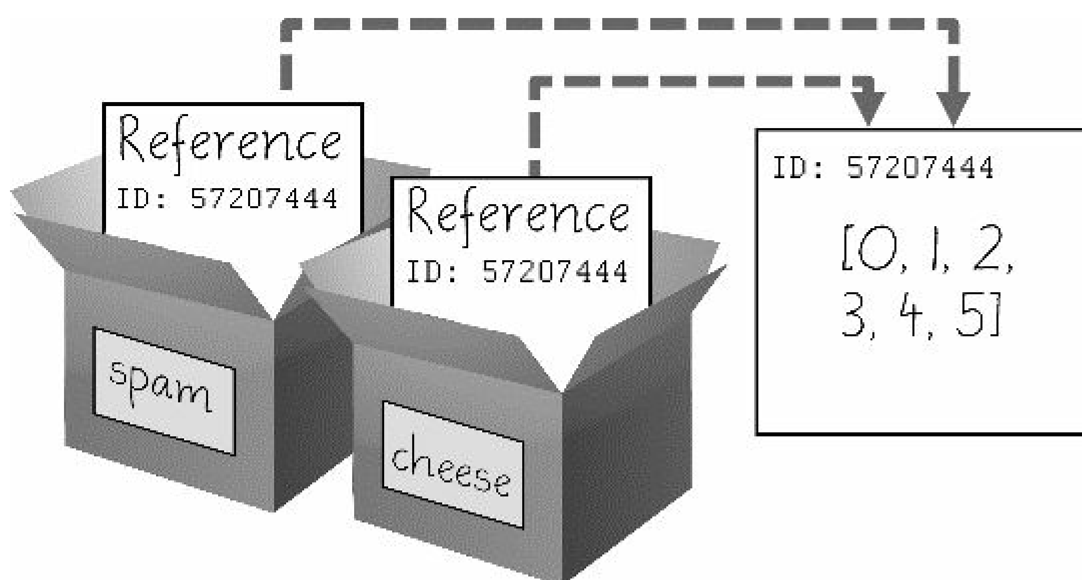


图4-5 spam = cheese复制了引用，而非列表

当你改变cheese指向的列表时，spam指向的列表也发生了改变，因为cheese和spam都指向同一个列表，如图4-6所示。

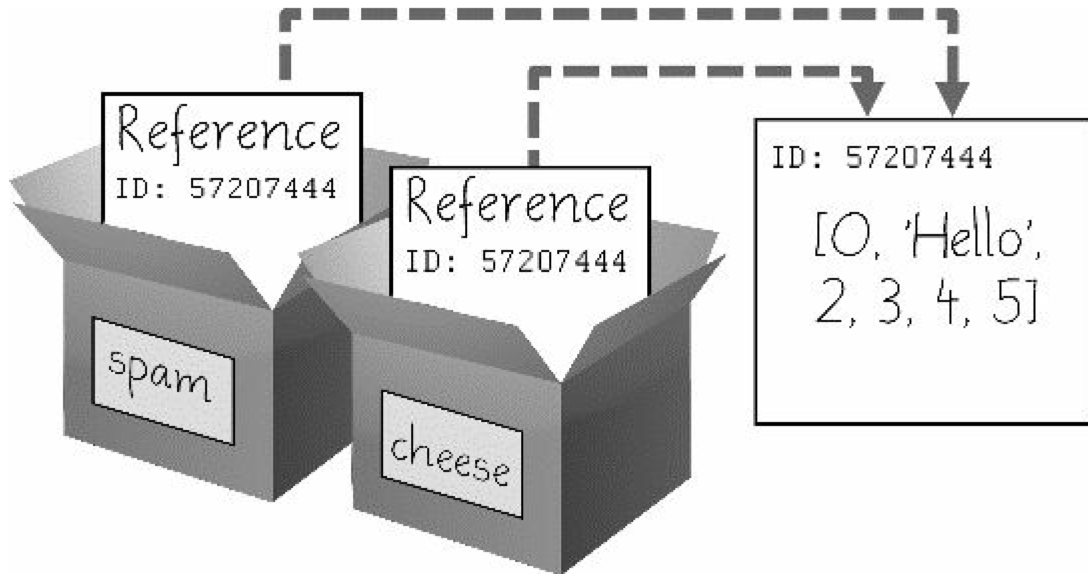


图4-6 cheese[1] = 'Hello!'修改了两个变量指向的列表

变量包含对列表值的引用，而不是列表值本身。但对于字符串和整数值，变量就包含了字符串或整数值。在变量必须保存可变数据类型的值时，例如列表或字典，Python就使用引用。对于不可变的数据类型的值，例如字符串、整型或元组，Python变量就保存值本身。

虽然Python变量在技术上包含了对列表或字典值的引用，但人们通常随意地说，该变量包含了列表或字典。

4.7.1 传递引用

要理解参数如何传递给函数，引用就特别重要。当函数被调用时，参数的值被复制给变元。对于列表（以及字典，我将在下一章中讨论），这意味着变元得到的是引用的拷贝。要看看这导致的后果，请打开一个新的文件编辑器窗口，输入以下代码，并保存为passingReference.py：

```
def eggs(someParameter):
    someParameter.append('Hello')
```

```
spam = [1, 2, 3]
eggs(spam)
print(spam)
```

请注意，当eggs()被调用时，没有使用返回值来为spam赋新值。相反，它直接当场修改了该列表。在运行时，该程序产生输出如下：

```
[1, 2, 3, 'Hello']
```

尽管spam和someParameter包含了不同的引用，但它们都指向相同的列表。这就是为什么函数内的append('Hello')方法调用在函数调用返回后，仍然会对该列表产生影响。

请记住这种行为：如果忘了Python处理列表和字典变量时采用这种方式，可能会导致令人困惑的缺陷。

4.7.2 copy模块的copy()和deepcopy()函数

在处理列表和字典时，尽管传递引用常常是最方便的方法，但如果函数修改了传入的列表或字典，你可能不希望这些变动影响原来的列表或字典。要做到这一点，Python提供了名为copy的模块，其中包含copy()和deepcopy()函数。第一个函数copy.copy()，可以用来复制列表或字典这样的可变值，而不只是复制引用。在交互式环境中输入以下代码：

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
```

```
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

现在spam和cheese变量指向独立的列表，这就是为什么当你将42赋给下标7时，只有cheese中的列表被改变。在图4-7中可以看到，两个变量的引用ID数字不再一样，因为它们指向了独立的列表。

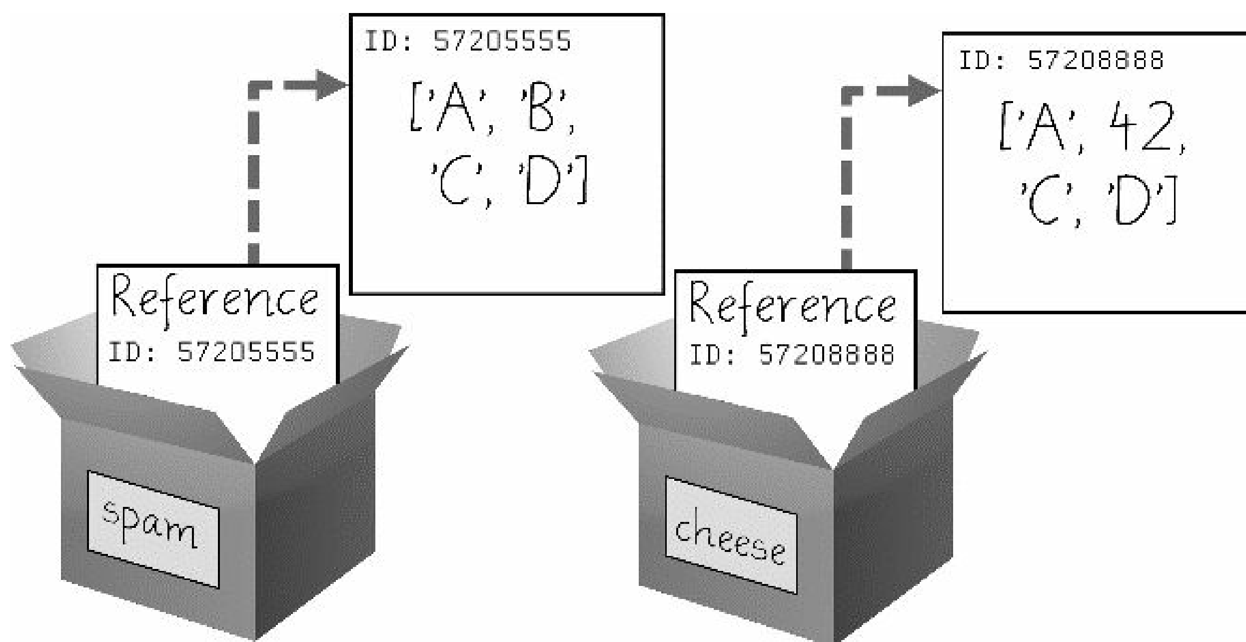


图4-7 cheese = copy.copy(spam)创建了第二个列表，能独立于第一个列表修改

如果要复制的列表中包含了列表，那就使用copy.deepcopy()函数来代替。deepcopy()函数将同时复制它们内部的列表。

4.8 小结

列表是有用的数据类型，因为它们让你写代码处理一组可以修改的值，同时仅用一个变量。在本书后面的章节中，你会看到一些程序利用列表来完成工作。没有列表，这些工作很困难，甚至不可能完成。

列表是可变的，这意味着它们的内容可以改变。元组和字符串虽然在某些方面像列表，却是不可变的，不能被修改。包含一个元组或字符串的变量，可以被一个新的元组或字符串覆写，但这和现场修改原来的值不是一回事，不像append()和remove()方法在列表上的效果。

变量不直接保存列表值，它们保存对列表的“引用”。在复制变量或将列表作为函数调用的参数时，这一点很重要。因为被复制的只是列表引用，所以要注意，对该列表的所有改动都可能影响到程序中的其他变量。如果需要对一个变量中的列表修改，同时不修改原来的列表，就可以用copy()或deepcopy()。

4.9 习题

1．什么是[]？

2．如何将'hello'赋给列表的第三个值，而列表保存在名为spam的变量中？（假定变量包含[2, 4, 6, 8, 10]）。

对接下来的3个问题，假定spam包含列表['a', 'b', 'c', 'd']。

3．spam[int('3' * 2) / 11]求值为多少？

4．spam[-1]求值为多少？

5．spam[:2]求值为多少？

对接下来的3个问题。假定bacon包含列表[3.14, 'cat', 11, 'cat', True]。

6．bacon.index('cat')求值为多少？

7．bacon.append(99)让bacon中的列表值变成什么样？

8．bacon.remove('cat')让bacon中的列表时变成什么样？

9．列表连接和复制的操作符是什么？

10．append()和insert()列表方法之间的区别是什么？

11．从列表中删除值有哪两种方法？

12．请说出列表值和字符串的几点相似之处。

13．列表和元组之间的区别是什么？

14．如果元组中只有一个整数值42，如何输入该元组？

15．如何从列表值得到元组形式？如何从元组值得到列表形式？

16．“包含”列表的变量，实际上并未真地直接包含列表。它们包含

的是什么？

17．copy.copy()和copy.deepcopy()之间的区别是什么？

4.10 实践项目

作为实践，编程完成下列任务。

4.10.1 逗号代码

假定有下面这样的列表：

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

编写一个函数，它以一个列表值作为参数，返回一个字符串。该字符串包含所有表项，表项之间以逗号和空格分隔，并在最后一个表项之前插入and。例如，将前面的spam列表传递给函数，将返回'apples, bananas, tofu, and cats'。但你的函数应该能够处理传递给它的任何列表。

4.10.2 字符图网格

假定有一个列表的列表，内层列表的每个值都是包含一个字符的字符串，像这样：

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

你可以认为grid[x][y]是一幅“图”在x、y坐标处的字符，该图由文本字符组成。原点(0, 0)在左上角，向右x坐标增加，向下y坐标增加。

复制前面的网格值，编写代码用它打印出图像。

```
..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....
```

提示

你需要使用循环嵌套循环，打印出grid[0][0]，然后grid[1][0]，然后grid[2][0]，以此类推，直到grid[8][0]。这就完成第一行，所以接下来打印换行。然后程序将打印出grid[0][1]，然后grid[1][1]，然后grid[2][1]，以此类推。程序最后将打印出grid[8][5]。

而且，如果你不希望在每次print()调用后都自动打印换行，记得向print()传递end关键字参数。

第5章　字典和结构化数据

在本章中，我将介绍字典数据类型，它提供了一种灵活的访问和组织数据的方式。然后，结合字典与前一章中关于列表的知识，你将学习如何创建一个数据结构，对井字棋盘建模。

5.1 字典数据类型

像列表一样，“字典”是许多值的集合。但不像列表的下标，字典的索引可以使用许多不同数据类型，不只是整数。字典的索引被称为“键”，键及其关联的值称为“键-值”对。

在代码中，字典输入时带花括号{}。在交互式环境中输入以下代码：

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

这将一个字典赋给myCat变量。这个字典的键是'size'、'color'和'disposition'。这些键相应的值是'fat'、'gray'和'loud'。可以通过它们的键访问这些值：

```
>>> myCat['size']
'fat'
```

```
>>> 'My cat has ' + myCat['color'] + ' fur.'

'My cat has gray fur.'
```

字典仍然可以用整数值作为键，就像列表使用整数值作为下标一样，但它们不必从0开始，可以是任何数字。

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

5.1.1 字典与列表

不像列表，字典中的表项是不排序的。名为spam的列表中，第一个表项是spam[0]。但字典中没有“第一个”表项。虽然确定两个列表是否相同时，表项的顺序很重要，但在字典中，键-值对输入的顺序并不重要。在交互式环境中输入以下代码：

```
>>> spam = ['cats', 'dogs', 'moose']

>>> bacon = ['dogs', 'moose', 'cats']
```

```
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

因为字典是不排序的，所以不能像列表那样切片。

尝试访问字典中不存在的键，将导致KeyError出错信息。这很像列表的“越界”IndexError出错信息。在交互式环境中输入以下代码，并注意显示的出错信息，因为没有'color'键：

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
>>> spam['color']
Traceback (most recent call last):
  File "", line 1, in
    spam['color']
KeyError: 'color'
```

尽管字典是不排序的，但可以用任意值作为键，这一点让你能够用强大的方式来组织数据。假定你希望程序保存朋友生日的数据，就可以使用一个字典，用名字作为键，用生日作为值。打开一个新的文件编辑窗口，输入以下代码，并保存为birthdays.py：

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

你创建了一个初始的字典，将它保存在birthdays中❶。用in关键字，可以看看输入的名字是否作为键存在于字典中❷，就像查看列表一样。如果该名字在字典中，你可以用方括号访问关联的值❸。如果不在，你可以用同样的方括号语法和赋值操作符添加它❹。

运行这个程序，结果看起来如下所示：

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

当然，在程序终止时，你在这个程序中输入的所有数据都丢失了。在第 8章中，你将学习如何将数据保存在硬盘的文件中。

5.1.2 keys()、values()和items()方法

有3个字典方法，它们将返回类似列表的值，分别对应于字典的键、值和键-值对：keys()、values()和items()。这些方法返回的值不是真正的列表，它们不能被修改，没有append()方法。但这些数据类型（分别是dict_keys、dict_values和dict_items）可以用于for循环。为了看看这些方法的工作原理，请在交互式环境中输入以下代码：

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

这里，for循环迭代了spam字典中的每个值。for循环也可以迭代每

个键，或者键-值对：

```
>>> for k in spam.keys():
        print(k)
color
age
>>> for i in spam.items():
        print(i)
('color', 'red')
('age', 42)
```

利用keys()、values()和items()方法，循环分别可以迭代键、值或键-值对。请注意，items()方法返回的dict_items值中，包含的是键和值的元组。

如果希望通过这些方法得到一个真正的列表，就把类似列表的返回值传递给 list函数。在交互式环境中输入以下代码：

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

list(spam.keys())代码行接受keys()函数返回的dict_keys值，并传递给list()。然后返回一个列表，即['color', 'age']。

也可以利用多重赋值的技巧，在for循环中将键和值赋给不同的变量。在交互式环境中输入以下代码：

```
>>> spam = {'color': 'red', 'age': 42}
```

```
>>> for k, v in spam.items():

        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

5.1.3 检查字典中是否存在键或值

回忆一下，前一章提到，in和not in操作符可以检查值是否存在于列表中。也可以利用这些操作符，检查某个键或值是否存在于字典中。在交互式环境中输入以下代码：

```
>>> spam = {'name': 'Zophie', 'age': 7}

>>> 'name' in spam.keys()
```

```
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

请注意，在前面的例子中，'color' in spam本质上是一个简写版本。相当于'color' in spam.keys()。这种情况总是对的：如果想要检查一个值是否为字典中的键，就可以用关键字in（或not in），作用于该字典本身。

5.1.4 get()方法

在访问一个键的值之前，检查该键是否存在于字典中，这很麻烦。

好在，字典有一个get()方法，它有两个参数：要取得其值的键，以及如果该键不存在时，返回的备用值。

在交互式环境中输入以下代码：

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

因为picnicItems字典中没有'egg'键，get()方法返回的默认值是0。不使用get()，代码就会产生一个错误消息，就像下面的例子：

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

```
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

5.1.5 setdefault()方法

你常常需要为字典中某个键设置一个默认值，当该键没有任何值时使用它。代码看起来像这样：

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

setdefault()方法提供了一种方式，在一行中完成这件事。传递给该方法的第一个参数，是要检查的键。第二个参数，是如果该键不存在时要设置的值。如果该键确实存在，方法就会返回键的值。在交互式环境中输入以下代码：

```
>>> spam = {'name': 'Pooka', 'age': 5}

>>> spam.setdefault('color', 'black')

'black'
```

```
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

第一次调用setdefault()时，spam变量中的字典变为{'color': 'black', 'age': 5, 'name': 'Pooka'}。该方法返回值'black'，因为现在该值被赋给键'color'。当spam.setdefault('color', 'white')接下来被调用时，该键的值“没有”被改变成'white'，因为spam变量已经有名为'color'的键。

setdefault()方法是一个很好的快捷方式，可以确保一个键存在。下面有一个小程序，计算一个字符串中每个字符出现的次数。打开一个文件编辑器窗口，输入以下代码，保存为characterCount.py：

```
message = 'It was a bright cold day in April, and the clocks were striking
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

程序循环迭代message字符串中的每个字符，计算每个字符出现的次数。setdefault()方法调用确保了键存在于count字典中（默认值是0），这样在执行count[character] = count[character] + 1时，就不会抛出KeyError错误。程序运行时，输出如下：

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, '
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6,
```

从输出可以看到，小写字母c出现了3次，空格字符出现了13次，大写字母A出现了1次。无论message变量中包含什么样的字符串，这个程序都能工作，即使该字符串有上百万的字符！

5.2 漂亮打印

如果程序中导入pprint模块，就可以使用pprint()和pformat()函数，它们将“漂亮打印”一个字典的字。如果想要字典中表项的显示比print()的输出结果更干净，这就有用了。修改前面的characterCount.py程序，将它保存为prettyCharacterCount.py。

```
import pprint





message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1
```

```
pprint



.pprint(



count)
```

这一次，当程序运行时，输出看起来更干净，键排过序。

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
```

```
 'w': 2,
 'y': 1}
```

如果字典本身包含嵌套的列表或字典，pprint.pprint()函数就特别有用。

如果希望得到漂亮打印的文本作为字符串，而不是显示在屏幕上，那就调用pprint.pformat()。下面两行代码是等价的：

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

5.3 使用数据结构对真实世界建模

甚至在因特网之前，人们也有办法与世界另一边的某人下一盘国际象棋。每个棋手在自己家里放好一个棋盘，然后轮流向对方寄出明信片，描述每一着棋。要做到这一点，棋手需要一种方法，无二义地描述棋盘的状态，以及他们的着法。

在“代数记谱法”中，棋盘空间由一个数字和字母坐标确定，如图5-1所示。

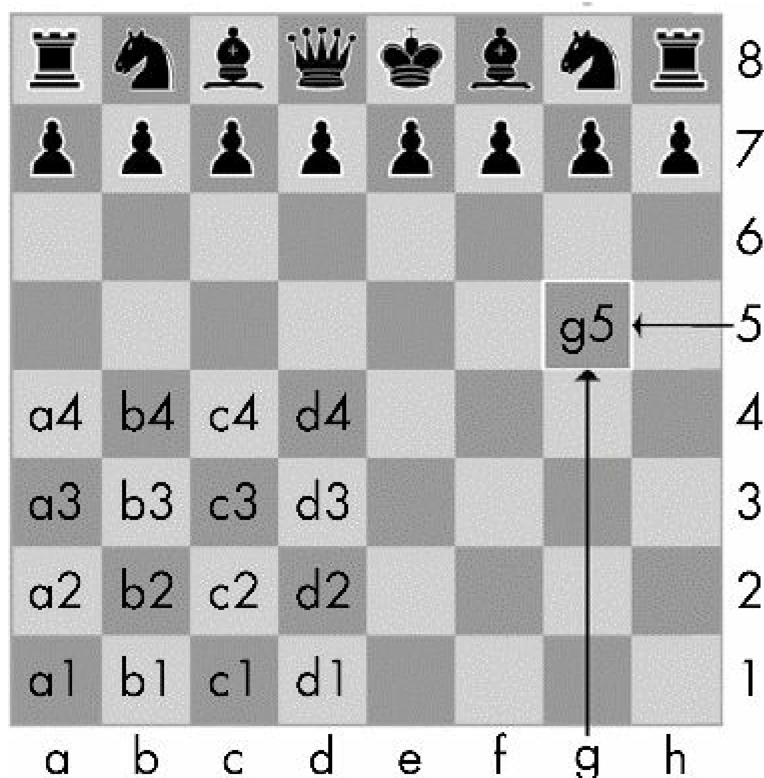


图5-1 代数记谱法中棋盘的坐标

棋子用字母表示：K表示王，Q表示后，R表示车，B表示象，N表示马。描述一次移动，用棋子的字母和它的目的地坐标。一对这样的移动表示一个回合（白方先下），例如，棋谱2. Nf3 Nc6表明在棋局的第二回合，白方将马移动到f3，黑方将马移动到c6。

代数记谱法还有更多内容，但要点是你可以用它无二义地描述象棋游戏，不需要站在棋盘前。你的对手甚至可以在世界的另一边！实际上，如果你的记忆力很好，甚至不需要物理的棋具：只需要阅读寄来的棋子移动，更新心里想的棋盘。

计算机有很好的记忆力。现在计算机上的程序，很容易存储几百万个像'2. Nf3 Nc6'这样的字符串。这就是为什么计算机不用物理棋盘就能下象棋。它们用数据建模来表示棋盘，你可以编写代码来使用这个模型。

这里就可以用到列表和字典。可以用它们对真实世界建模，例如棋盘。作为第一个例子，我们将使用比国际象棋简单一点的游戏：井字棋。

5.3.1 井字棋盘

井字棋盘看起来像一个大的井字符号（#），有9个空格，可以包含X、O或空。要用字典表示棋盘，可以为每个空格分配一个字符串键，如图5-2所示。

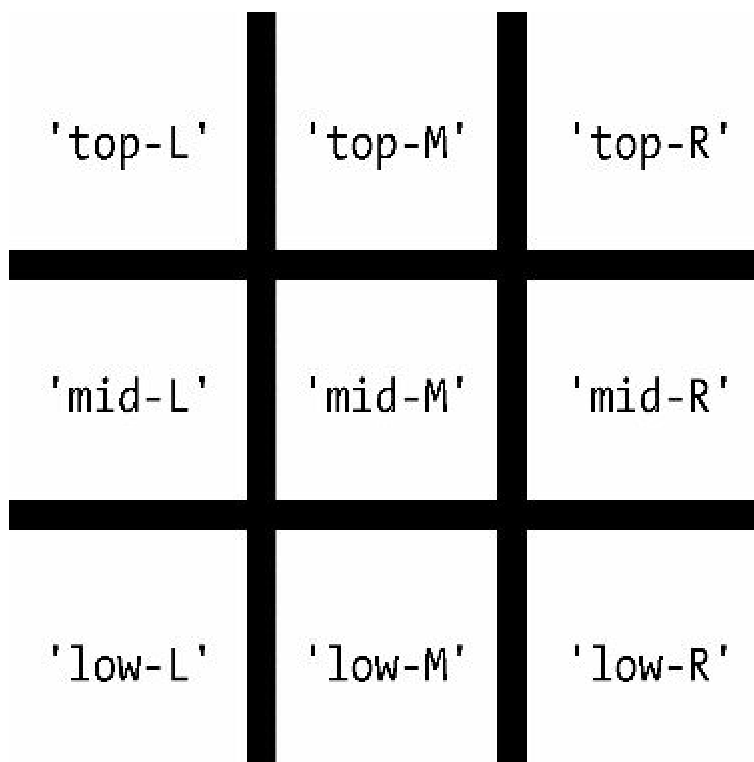
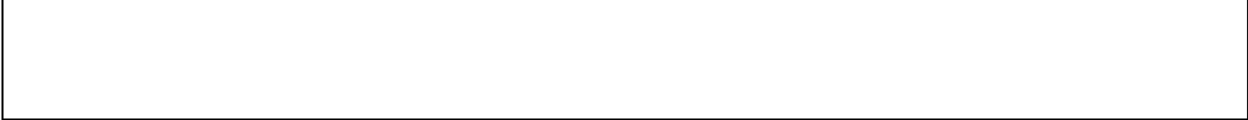


图5-2 井字棋盘的空格和它们对应的键

可以用字符串值来表示，棋盘上每个空格有什么：'X'、'O'或' '（空格字符）。因此，需要存储9个字符串。可以用一个字典来做这事。带有键'top-R'的字符串表示右上角，带有键'low-L'的字符串表示左下角，带有键'mid-M'的字符串表示中间，以此类推。

这个字典就是表示井字棋盘的数据结构。将这个字典表示的棋盘保存在名为theBoard的变量中。打开一个文件编辑器窗口，输入以下代码，并保存为ticTacToe.py：

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```



保存在theBoard变量中的数据结构，表示了图5-3中的井字棋盘。

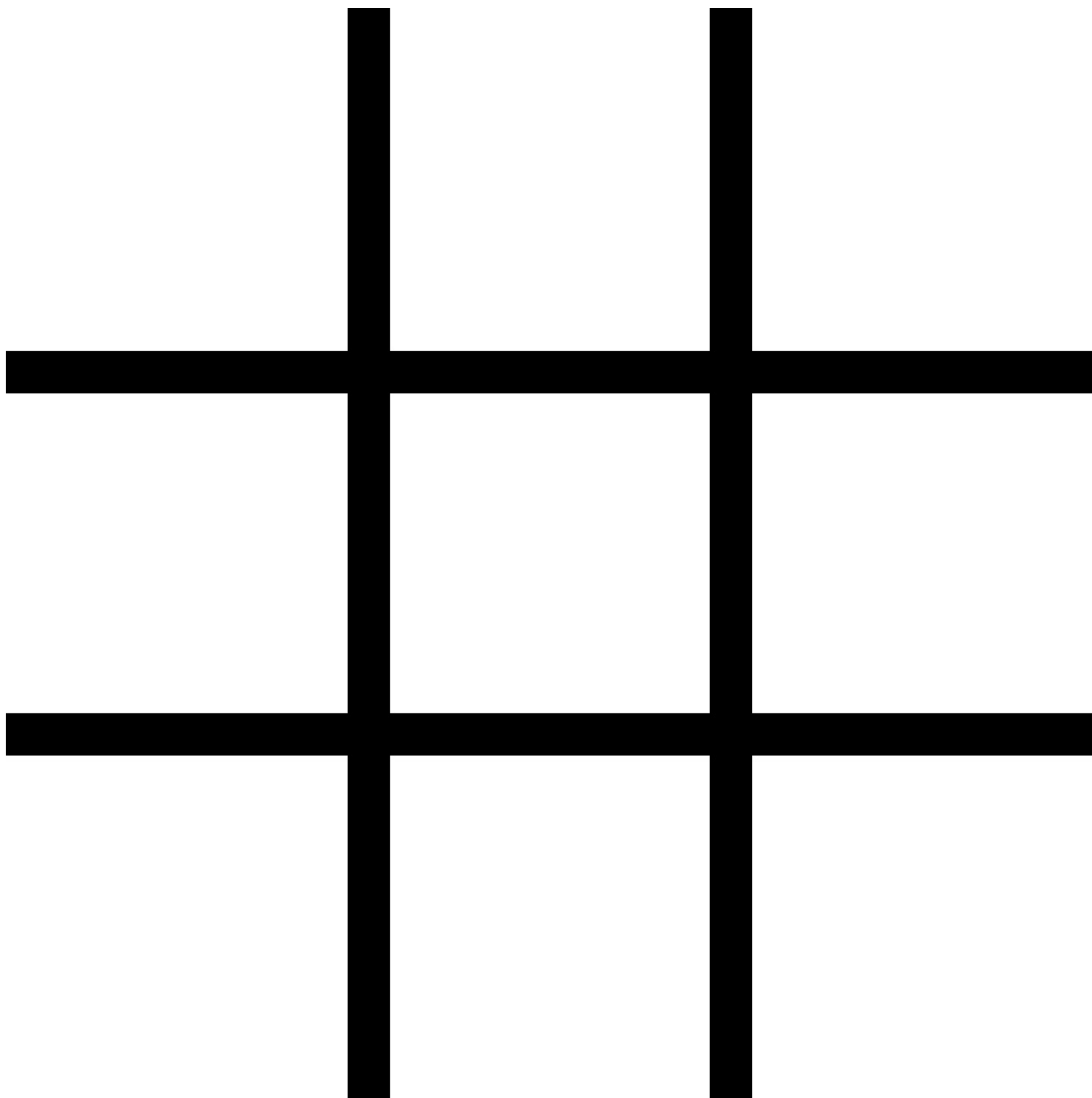
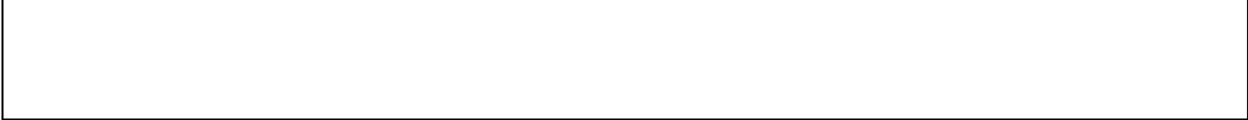


图5-3 一个空的井字棋盘

因为theBoard变量中每个键的值都是单个空格字符，所以这个字典表示一个完全干净的棋盘。如果玩家X选择了中间的空格，就可以用下面这个字典来表示棋盘：

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```



theBoard变量中的数据结构现在表示图5-4中的井字棋盘。

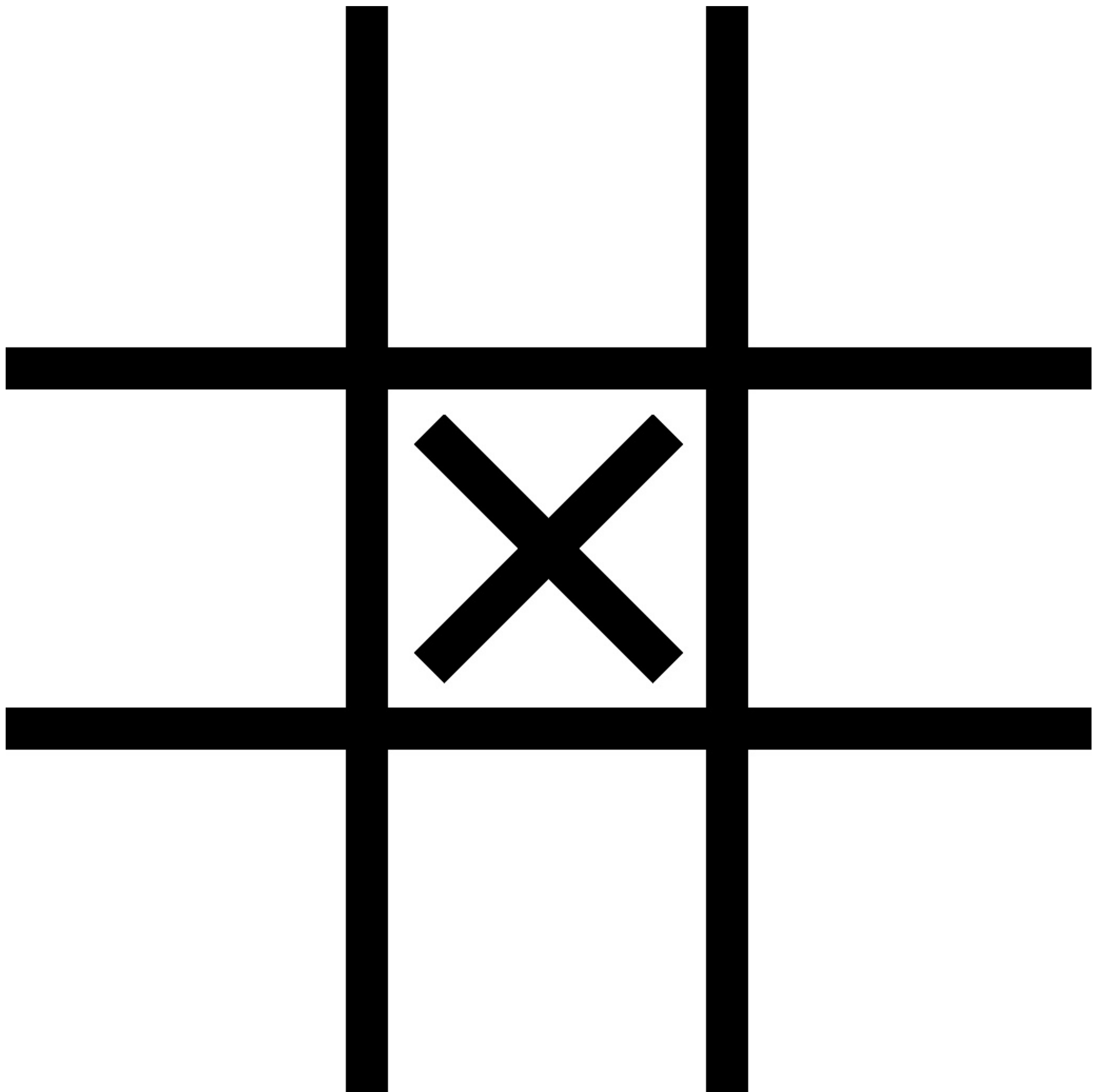


图5-4 第一着

一个玩家O获胜的棋盘上，他将O横贯棋盘的顶部，看起来像这样：

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

theBoard变量中的数据结构现在表示图5-5中的井字棋盘。

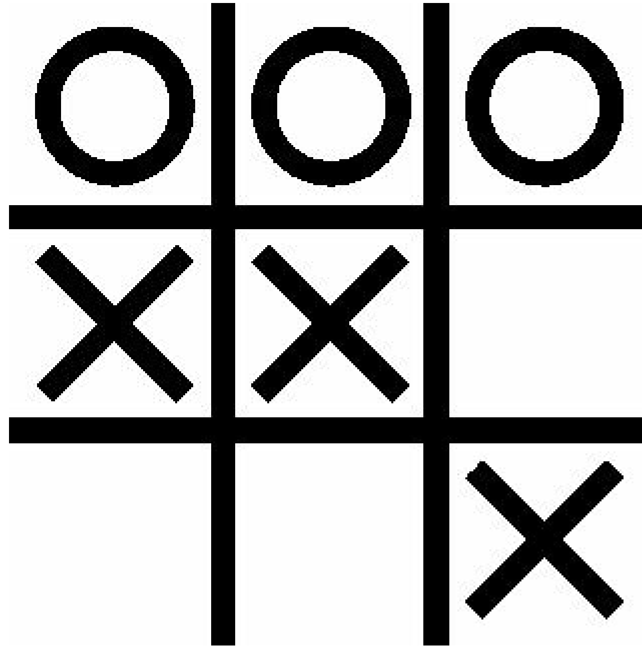


图5-5 玩家O获胜

当然，玩家只看到打印在屏幕上的内容，而不是变量的内容。让我们创建一个函数，将棋盘字典打印到屏幕上。将下面代码添加到ticTacToe.py（新代码是黑体的）：

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
```

```
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])

    print('-+-+-')

    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

printBoard(theBoard)
```

运行这个程序时，printBoard()将打印出空白井字棋盘。

```
| |
-+-+-
| |
-+-+-
| |
```

printBoard()函数可以处理传入的任何井字棋数据结构。尝试将代码改成以下的样子：

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M'

'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):

print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])

print('-+-+-')

print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])

print('-+-+-')
```

```
print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

现在运行该程序，新棋盘将打印在屏幕上。

```
O|O|O
-+-+-
X|X|
-+-+-
| |X
```

因为你创建了一个数据结构来表示井字棋盘，编写了printBoard()中的代码来解释该数据结构，所以就有了一个程序，对井字棋盘进行了“建模”。也可以用不同的方式组织数据结构（例如，使用'TOP-LEFT'这样的键来代替'top-L'），但只要代码能处理你的数据结构，就有了正确工作的程序。

例如，printBoard()函数预期井字棋数据结构是一个字典，包含所有9个空格的键。假如传入的字典缺少'mid-L'键，程序就不能工作了。

```
O|O|O
-+-+-
Traceback (most recent call last):
  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)
  File "ticTacToe.py", line 6, in printBoard
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

现在让我们添加代码，允许玩家输入他们的着法。修改ticTacToe.py程序如下所示：

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶     printBoard(theBoard)
```

```
    print('Turn for ' + turn + '. Move on which space?')
❷     move = input()
❸     theBoard[move] = turn
❹     if turn == 'X':
          turn = 'O'
      else:
          turn = 'X'
printBoard(theBoard)
```

新的代码在每一步新的着法之前，打印出棋盘❶，获取当前棋手的着法❷，相应地更新棋盘❸，然后改变当前棋手❹，进入到下一着。

运行该程序，它看起来像这样：

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M




 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L




 | |
-+-+-
 |X|
-+-+-
O| |

--_snip_--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
```

```
Turn for X. Move on which space?
low-M





O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

这不是一个完整的井字棋游戏（例如，它并不检查玩家是否获胜），但这已足够展示如何在程序中使用数据结构。

> **注意**
>
> 如果你很好奇，完整的井字棋程序的源代码在网上有介绍，网址是http://nostarch.com/automatestuff/ 。

5.3.2 嵌套的字典和列表

对井字棋盘建模相当简单：棋盘只需要一个字典，包含9个键值对。当你对复杂的事物建模时，可能发现字典和列表中需要包含其他字典和列表。列表适用于包含一组有序的值，字典适合于包含关联的键与值。例如，下面的程序使用字典包含其他字典，用于记录谁为野餐带来了什么食物。totalBrought()函数可以读取这个数据结构，计算所有客人带来的食物的总数。

```
  allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
               'Bob': {'ham sandwiches': 3, 'apples': 2},
               'Carol': {'cups': 3, 'apple pies': 1}}

  def totalBrought(guests, item):
      numBrought = 0
❶     for k, v in guests.items():
❷         numBrought = numBrought + v.get(item, 0)
      return numBrought
```

```
print('Number of things being brought:')
print(' - Apples ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches'
print(' - Apple Pies ' + str(totalBrought(allGuests, 'apple pies')))
```

在totalBrought()函数中，for循环迭代guests中的每个键值对❶。在这个循环里，客人的名字字符串赋给k，他们带来的野餐食物的字典赋给v。如果食物参数是字典中存在的键，它的值（数量）就添加到numBrought❷。如果它不是键，get()方法就返回0，添加到numBrought。

该程序的输出像这样：

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1
```

这似乎对一个非常简单的东西建模，你可能认为不需要费事去写一个程序来做到这一点。但是要认识到，这个函数totalBrought()可以轻易地处理一个字典，其中包含数千名客人，每个人都带来了“数千种”不同的野餐食物。这样用这种数据结构来保存信息，并使用totalBrought()函数，就会节约大量的时间！

你可以用自己喜欢的任何方法，用数据结构对事物建模，只要程序中其他代码能够正确处理这个数据模型。在刚开始编程时，不需要太担心数据建模的“正确”方式。随着经验增加，你可能会得到更有效的模型，但重要的是，该数据模型符合程序的需要。

5.4 小结

在本章中，你学习了字典的所有相关知识。列表和字典是这样的值，它们可以包含多个值，包括其他列表和字典。字典是有用的，因为你可以把一些项（键）映射到另一些项（值），它不像列表，只包含一系列有序的值。字典中的值是通过方括号访问的，像列表一样。字典不是只能使用整数下标，而是可以用各种数据类型作为键：整型、浮点型、字符串或元组。通过将程序中的值组织成数据结构，你可以创建真实世界事物的模型。井字棋盘就是这样一个例子。

这就介绍了Python编程的所有基本概念！在本书后面的部分，你将继续学习一些新概念，但现在你已学习了足够多的内容，可以开始编写一些有用的程序，让一些任务自动化。你可能不觉得自己有足够的Python知识，来实现页面下载、更新电子表格，或发送文本消息。但这就是Python模块要干的事！这些模块由其他程序员编写，提供了一些函数，让这些事情变得容易。所以让我们学习如何编写真正的程序，实现有用的自动化任务。

5.5 习题

1．空字典的代码是怎样的？

2．一个字典包含键'fow'和值42，看起来是怎样的？

3．字典和列表的主要区别是什么？

4．如果spam是{'bar': 100}，你试图访问spam['foo']，会发生什么？

5．如果一个字典保存在spam中，表达式'cat' in spam和'cat' in spam.keys()之间的区别是什么？

6．如果一个字典保存在变量中，表达式'cat' in spam和'cat' in spam.values()之间的区别是什么？

7．下面代码的简洁写法是什么？

```
if 'color' not in spam:
```

```
spam['color'] = 'black'
```

8．什么模块和函数可以用于“漂亮打印”字典值？

5.6 实践项目

作为实践，编程完成下列任务。

5.6.1 好玩游戏的物品清单

你在创建一个好玩的视频游戏。用于对玩家物品清单建模的数据结构是一个字典。其中键是字符串，描述清单中的物品，值是一个整型值，说明玩家有多少该物品。例如，字典值{'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}意味着玩家有1条绳索、6个火把、42枚金币等。

写一个名为displayInventory()的函数，它接受任何可能的物品清单，并显示如下：

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
1 dagger
Total number of items: 62
```

> **提示**
>
> 你可以使用for循环，遍历字典中所有的键。

```
# inventory.py
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}
```

```
def displayInventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        print(str(v) + ' ' + k)
        item_total += v
    print("Total number of items: " + str(item_total))

displayInventory(stuff)
```

5.6.2 列表到字典的函数，针对好玩游戏物品清单

假设征服一条龙的战利品表示为这样的字符串列表：

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

写一个名为addToInventory(inventory, addedItems)的函数，其中inventory参数是一个字典，表示玩家的物品清单（像前面项目一样），addedItems参数是一个列表，就像dragonLoot。

addToInventory()函数应该返回一个字典，表示更新过的物品清单。请注意，列表可以包含多个同样的项。你的代码看起来可能像这样：

```
def addToInventory(inventory, addedItems):
    # your code goes here

inv = {'gold coin': 42, 'rope': 1}
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = addToInventory(inv, dragonLoot)
displayInventory(inv)
```

前面的程序（加上前一个项目中的displayInventory()函数）将输出如下：

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger

Total number of items: 48
```

第6章　字符串操作

文本是程序需要处理的最常见的数据形式。你已经知道如何用+操作符连接两个字符串，但能做的事情还要多得多。可以从字符串中提取部分字符串，添加或删除空白字符，将字母转换成小写或大写，检查字符串的格式是否正确。你甚至可以编写Python代码访问剪贴板，复制或粘贴文本。

在本章中，你将学习所有这些内容和更多内容。然后你会看到两个不同的编程项目：一个是简单的口令管理器，另一个将枯燥的文本格式化工作自动化。

6.1 处理字符串

让我们来看看，Python提供的写入、打印和访问字符串的一些方法。

6.1.1 字符串字面量

在Python中输入字符串值相当简单的：它们以单引号开始和结束。但是如何才能在字符串内使用单引号呢？输入'That is Alice's cat.'是不行的，因为Python认为这个字符串在Alice之后就结束了，剩下的（s cat.'）是无效的Python代码。好在，有几种方法来输入字符串。

6.1.2 双引号

字符串可以用双引号开始和结束，就像用单引号一样。使用双引号的一个好处，就是字符串中可以使用单引号字符。在交互式环境中输入以下代码：

```
>>> spam
```

```
 = "That is Alice's cat."
```

因为字符串以双引号开始，所以Python知道单引号是字符串的一部分，而不是表示字符串的结束。但是，如果在字符串中既需要使用单引号又需要使用双引号，那就要使用转义字符。

6.1.3 转义字符

“转义字符”让你输入一些字符，它们用其他方式是不可能放在字符串里的。转义字符包含一个倒斜杠（\），紧跟着是想要添加到字符串中的字符。（尽管它包含两个字符，但大家公认它是一个转义字符。）例如，单引号的转义字符是\’。你可以在单引号开始和结束的字符串中使用它。为了看看转义字符的效果，在交互式环境中输入以下代码：

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python知道，因为Bob\'s中的单引号有一个倒斜杠，所以它不是表示字符串结束的单引号。转义字符\'和\"让你能在字符串中加入单引号和双引号。

表6-1列出了可用的转义字符。

表**6-1** 转义字符

转义字符	打印为
\'	单引号
\"	双引号
\t	制表符
\n	换行符
\\	倒斜杠

在交互式环境中输入以下代码：

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")



Hello there!
How are you?
I'm doing fine.
```

6.1.4 原始字符串

可以在字符串开始的引号之前加上r，使它成为原始字符串。“原始字符串”完全忽略所有的转义字符，打印出字符串中所有的倒斜杠。例如，在交互式环境中输入以下代码：

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

因为这是原始字符串，Python认为倒斜杠是字符串的一部分，而不是转义字符的开始。如果输入的字符串包含许多倒斜杠，比如下一章中要介绍的正则表达式字符串，那么原始字符串就很有用。

6.1.5 用三重引号的多行字符串

虽然可以用\n转义字符将换行放入一个字符串，但使用多行字符串通常更容易。在Python中，多行字符串的起止是3个单引号或3个双引号。“三重引号”之间的所有引号、制表符或换行，都被认为是字符串的一部分。Python的代码块缩进规则不适用于多行字符串。

打开文件编辑器，输入以下代码：

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

将该程序保存为catnapping.py并运行。输出看起来像这样：

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.
```

```
Sincerely,
Bob
```

请注意，Eve's中的单引号字符不需要转义。在原始字符串中，转义单引号和双引号是可选的。下面的print()调用将打印出同样的文本，但没有使用多行字符串：

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat burglary, and extortion.\n\nSincerely,\nBob')
```

6.1.6 多行注释

虽然井号字符（#）表示这一行是注释，但多行字符串常常用作多行注释。下面是完全有效的Python代码：

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

6.1.7 字符串下标和切片

字符串像列表一样，使用下标和切片。可以将字符串'Hello

world!'看成是一个列表，字符串中的每个字符都是一个表项，有对应的下标。

```
'  H  e  l  l  o  w  o  r  l  d  !  '
    0  1  2  3  4  5  6  7  8  9  10 11
```

字符计数包含了空格和感叹号，所以'Hello world!'有12个字符，H的下标是0，!的下标是11。在交互式环境中输入以下代码：

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
```

```
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

如果指定一个下标，你将得到字符串在该处的字符。如果用一个下标和另一个下标指定一个范围，开始下标将被包含，结束下标则不包含。因此，如果spam是'Hello world!'，spam[0:5]就是'Hello'。通过spam[0:5]得到的子字符串，将包含spam[0]到spam[4]的全部内容，而不包括下标5处的空格。

请注意，字符串切片并没有修改原来的字符串。可以从一个变量中获取切片，记录在另一个变量中。在交互式环境中输入以下代码：

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
```

```
'Hello'
```

通过切片并将结果子字符串保存在另一个变量中，就可以同时拥有完整的字符串和子字符串，便于快速简单的访问。

6.1.8 字符串的in和not in操作符

像列表一样，in和not in操作符也可以用于字符串。用in或not in连接两个字符串得到的表达式，将求值为布尔值True或False。在交互式环境中输入以下代码：

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>> '' in 'spam'
```

```
True
>>> 'cats' not in 'cats and dogs'
False
```

这些表达式测试第一个字符串（精确匹配，区分大小写）是否在第二个字符串中。

6.2 有用的字符串方法

一些字符串方法会分析字符串，或生成转变过的字符串。本节介绍了这些方法，你会经常使用它们。

6.2.1 字符串方法upper()、lower()、isupper()和islower()

upper()和lower()字符串方法返回一个新字符串，其中原字符串的所有字母都被相应地转换为大写或小写。字符串中非字母字符保持不变。

在交互式环境中输入以下代码：

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
```

```
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

请注意，这些方法没有改变字符串本身，而是返回一个新字符串。如果你希望改变原来的字符串，就必须在该字符串上调用upper()或lower()，然后将这个新字符串赋给保存原来字符串的变量。这就是为什么必须使用 spam = spam.upper()，才能改变spam中的字符串，而不是仅仅使用spam.upper()（这就好比，如果变量eggs中包含值10，写下eggs + 3并不会改变eggs的值，但是eggs = eggs + 3会改变egg的值）。

如果需要进行大小写无关的比较，upper()和lower()方法就很有用。字符串'great'和'GREat'彼此不相等。但在下面的小程序中，用户输入Great、GREAT或grEAT都没关系，因为字符串首先被转换成小写。

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
```

```
    print('I hope the rest of your day is good.')
```

在运行该程序时，先显示问题，然后输入变形的great，如GREat，程序将给出输出I feel great too。在程序中加入代码，处理多种用户输入情况或输入错误，诸如大小写不一致，这会让程序更容易使用，且更不容易失效。

```
How are you?
GREat
I feel great too.
```

如果字符串至少有一个字母，并且所有字母都是大写或小写，isupper()和islower()方法就会相应地返回布尔值True。否则，该方法返回False。在交互式环境中输入以下代码，并注意每个方法调用的返回值：

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
```

```
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

因为upper()和lower()字符串方法本身返回字符串，所以也可以在“那些”返回的字符串上继续调用字符串方法。这样做的表达式看起来就像方法调用链。在交互式环境中输入以下代码：

```
>>> 'Hello'.upper()
```

```
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

6.2.2 isX字符串方法

除了islower()和isupper()，还有几个字符串方法，它们的名字以is开始。这些方法返回一个布尔值，描述了字符串的特点。下面是一些常用的isX字符串方法：

- isalpha()返回True，如果字符串只包含字母，并且非空；

- isalnum()返回True，如果字符串只包含字母和数字，并且非空；
- isdecimal()返回True，如果字符串只包含数字字符，并且非空；
- isspace()返回True，如果字符串只包含空格、制表符和换行，并且非空；
- .istitle()返回True，如果字符串仅包含以大写字母开头、后面都是小写字母的单词。

在交互式环境中输入以下代码：

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
```

```
True
>>> 'This Is Title Case'.istitle()

True
>>> 'This Is Title Case 123'.istitle()

True
>>> 'This Is not Title Case'.istitle()

False
>>> 'This Is NOT Title Case Either'.istitle()

False
```

如果需要验证用户输入，isX字符串方法是有用的。例如，下面的程序反复询问用户年龄和口令，直到他们提供有效的输入。打开一个新的文件编辑器窗口，输入以下程序，保存为validateInput.py：

```
while True:
    print('Enter your age:')
```

```
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

在第一个while循环中，我们要求用户输入年龄，并将输入保存在age中。如果age是有效的值（数字），我们就跳出第一个while循环，转向第二个循环，询问口令。否则，我们告诉用户需要输入数字，并再次要求他们输入年龄。在第二个while循环中，我们要求输入口令，客户的输入保存在password中。如果输入是字母或数字，就跳出循环。如果不是，我们并不满意，于是告诉用户口令必须是字母或数字，并再次要求他们输入口令。

如果运行，该程序的输出看起来如下：

```
Enter your age:
forty two
Please enter a number for your age.
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
```

```
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

在变量上调用isdecimal()和isalnum()，我们就能够测试保存在这些变量中的值是否为数字，是否为字母或数字。这里，这些测试帮助我们拒绝输入forty two，接受42，拒绝secr3t!，接受secr3t。

6.2.3 字符串方法startswith()和endswith()

startswith()和endswith()方法返回True，如果它们所调用的字符串以该方法传入的字符串开始或结束。否则，方法返回False。在交互式环境中输入以下代码：

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
```

```
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

如果只需要检查字符串的开始或结束部分是否等于另一个字符串，而不是整个字符串，这些方法就可以替代等于操作符==，这很有用。

6.2.4 字符串方法join()和split()

如果有一个字符串列表，需要将它们连接起来，成为一个单独的字符串，join()方法就很有用。join()方法在一个字符串上调用，参数是一个字符串列表，返回一个字符串。返回的字符串由传入的列表中每个字符串连接而成。例如，在交互式环境中输入以下代码：

```
>>> ', '.join(['cats', 'rats', 'bats'])
```

```
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

请注意，调用join()方法的字符串，被插入到列表参数中每个字符串的中间。例如，如果在', '字符串上调用join(['cats', 'rats', 'bats'])，返回的字符串就是'cats, rats, bats'。

要记住，join()方法是针对一个字符串而调用的，并且传入一个列表值（很容易不小心用其他的方式调用它）。split()方法做的事情正好相反：它针对一个字符串调用，返回一个字符串列表。在交互式环境中输入以下代码：

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

默认情况下，字符串'My name is Simon'按照各种空白字符分割，诸如空格、制表符或换行符。这些空白字符不包含在返回列表的字符串中。也可以向split()方法传入一个分割字符串，指定它按照不同的字符串分割。例如，在交互式环境中输入以下代码：

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

一个常见的split()用法，是按照换行符分割多行字符串。在交互式环境中输入以下代码：

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
```

```
that is labeled "Milk Experiment".

Please do not drink it.

Sincerely,

Bob'''

>>> spam.split('\n')

['Dear Alice,', 'How have you been? I am fine.', 'There is a container in t
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.
'Sincerely,', 'Bob']
```

向split()方法传入参数'\n'，我们按照换行符分割变量中存储的多行字符串，返回列表中的每个表项，对应于字符串中的一行。

6.2.5 用**rjust()**、**ljust()**和**center()**方法对齐文本

rjust()和ljust()字符串方法返回调用它们的字符串的填充版本，通过插入空格来对齐文本。这两个方法的第一个参数是一个整数长度，用于对齐字符串。在交互式环境中输入以下代码：

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

'Hello'.rjust(10)是说我们希望右对齐，将'Hello'放在一个长度为10的字符串中。'Hello'有5个字符，所以左边会加上5个空格，得到一个10个字符的字符串，实现'Hello'右对齐。

rjust()和ljust()方法的第二个可选参数将指定一个填充字符，取代空格字符。在交互式环境中输入以下代码：

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

center()字符串方法与ljust()与rjust()类似，但它让文本居中，而不是左对齐或右对齐。在交互式环境中输入以下代码：

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

如果需要打印表格式数据，留出正确的空格，这些方法就特别有用。打开一个新的文件编辑器窗口，输入以下代码，并保存为picnicTable.py：

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

在这个程序中，我们定义了printPicnic()方法，它接受一个信息的字典，并利用center()、ljust()和rjust()，以一种干净对齐的表格形式显示这些信息。

我们传递给printPicnic()的字典是picnicItems。在picnicItems中，我们有4个三明治、12个苹果、4个杯子和8000块饼干。我们希望将这些信息组织成两行，表项的名字在左边，数量在右边。

要做到这一点，就需要决定左列和右列的宽度。与字典一起，我们将这些值传递给printPicnic()。

printPicnic()接受一个字典，一个leftWidth表示表的左列宽度，一个rightWidth表示表的右列宽度。它打印出标题PICNIC ITEMS，在表上方居中。然后它遍历字典，每行打印一个键-值对。键左对齐，填充句号。值右对齐，填充空格。

在定义printPicnic()后，我们定义了字典picnicItems，并调用printPicnic()两次，传入不同的表左右列宽度。

运行该程序，野餐用品就会显示两次。第一次左列宽度是12个字符，右列宽度是5个字符。第二次它们分别是20个和6个字符。

```
---PICNIC ITEMS--
sandwiches.. 4
apples...... 12
cups........ 4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.......... 4
apples.............. 12
cups................ 4
cookies............. 8000
```

利用rjust()、ljust()和center()让你确保字符串整齐对齐，即使你不清楚字符串有多少字符。

6.2.6 用**strip()**、**rstrip()**和**lstrip()**删除空白字符

有时候你希望删除字符串左边、右边或两边的空白字符（空格、制表符和换行符）。strip()字符串方法将返回一个新的字符串，它的开头或末尾都没有空白字符。lstrip()和rstrip()方法将相应删除左边或右边的空白字符。

在交互式环境中输入以下代码：

```
>>> spam = ' Hello World '
>>> spam.strip()
'Hello World'
```

```
>>> spam.lstrip()
'Hello World '
>>> spam.rstrip()
'     Hello World'
```

有一个可选的字符串参数，指定两边的哪些字符应该删除。在交互式环境中输入以下代码：

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

向strip()方法传入参数'ampS'，告诉它在变量中存储的字符串两端，删除出现的a、m、p和大写的S。传入strip()方法的字符串中，字符的顺序并不重要：strip('ampS')做的事情和strip('mapS')或strip('Spam')一样。

6.2.7 用pyperclip模块拷贝粘贴字符串

pyperclip模块有copy()和paste()函数，可以向计算机的剪贴板发送文本，或从它接收文本。将程序的输出发送到剪贴板，使它很容易粘贴到邮件、文字处理程序或其他软件中。pyperclip模块不是Python自带的。要安装它，请遵从附录A中安装第三方模块的指南。安装pyperclip模块后，在交互式环境中输入以下代码：

```
>>> import pyperclip

>>> pyperclip.copy('Hello world!')

>>> pyperclip.paste()

'Hello world!'
```

当然，如果你的程序之外的某个程序改变了剪贴板的内容，paste()函数就会返回它。例如，如果我将这句话复制到剪贴板，然后调用paste()，看起来就会像这样：

```
>>> pyperclip.paste()
```

```
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

在IDLE之外运行Python脚本

到目前为止，你一直在使用IDLE中的交互式环境和文件编辑器来运行Python脚本。但是，你不想每次运行一个脚本时，都打开IDLE和Python脚本，这样不方便。好在，有一些快捷方式，让你更容易地建立和运行Python脚本。这些步骤在Windows、OS X和Linux上稍有不同，但每一种都在附录B中描述。请翻到附录B，学习如何方便地运行Python脚本，并能够向它们传递命令行参数。（使用IDLE时，不能向程序传递命令行参数。）

6.3 项目：口令保管箱

你可能在许多不同网站上拥有账号，每个账号使用相同的口令是个坏习惯。如果这些网站中任何一个有安全漏洞，黑客就会知道你所有的其他账号的口令。最好是在你的计算机上，使用口令管理器软件，利用一个主控口令，解锁口令管理器。然后将某个账户口令拷贝到剪贴板，再将它粘贴到网站的口令输入框。

你在这个例子中创建的口令管理器程序并不安全，但它基本展示了这种程序的工作原理。

本章项目

这是本书的第一个章内项目。以后，每章都会有一些项目，展示该章介绍的一些概念。这些项目的编写方式，让你从一个空白的文件编辑器窗口开始，得到一个完整的、能工作的程序。就像交互式环境的例子一样，不要只看项目的部分，要注意计算机的提示！

第1步：程序设计和数据结构

你希望用一个命令行参数来运行这个程序，该参数是账号的名称。例如，账号的口令将拷贝到剪贴板，这样用户就能将它粘贴到口令输入框。通过这种方式，用户可以有很长而复杂的口令，又不需要记住它们。

打开一个新的文件编辑器窗口，将该程序保存为pw.py。程序开始时需要有一行#!（参见附录B），并且应该写一些注释，简单描述该程

序。因为你希望关联每个账号的名称及其口令，所以可以将这些作为字符串保存在字典中。字典将是组织你的账号和口令数据的数据结构。让你的程序看起来像下面这样：

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}
```

第2步：处理命令行参数

命令行参数将存储在变量sys.argv中（关于如何在程序中使用命令行参数，更多信息请参见附录B）。sys.argv列表中的第一项总是一个字符串，它包含程序的文件名（'pw.py'）。第二项应该是第一个命令行参数。对于这个程序，这个参数就是账户名称，你希望获取它的口令。因为命令行参数是必须的，所以如果用户忘记添加参数（也就是说，如果列表中少于两个值），你就显示用法信息。让你的程序看起来像下面这样：

```
#! python3
# pw.py - An insecure password locker program.

PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys



if len(sys.argv) < 2:
```

```
    print('Usage: python pw.py [account] - copy account password')

    sys.exit()

account = sys.argv[1] # first command line arg is the account name
```

第3步：复制正确的口令

既然账户名称已经作为字符串保存在变量account中，你就需要看看它是不是PASSWORDS字典中的键。如果是，你希望利用pyperclip.copy()，将该键的值复制到剪贴板（既然用到了pyperclip模块，就需要导入它）。请注意，实际上不需要account变量，你可以在程序中所有使用account的地方，直接使用sys.argv[1]。但名为account的变量更可读，不像是神秘的sys.argv[1]。

让你的程序看起来像这样：

```
#! python3
# pw.py - An insecure password locker program.
```

```
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',

'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',

'luggage': '12345'}

import sys, pyperclip

if len(sys.argv) < 2:
    print('Usage: py pw.py [account] - copy account password')
    sys.exit()

account = sys.argv[1] # first command line arg is the account name

if account in PASSWORDS:

    pyperclip.copy(PASSWORDS[account])

    print('Password for ' + account + ' copied to clipboard.')

else:
```

```
    print('There is no account named ' + account)
```

这段新代码在PASSWORDS字典中查找账户名称。如果该账号名称是字典中的键，我们就取得该键对应的值，将它复制到剪贴板，然后打印一条消息，说我们已经复制了该值。否则，我们打印一条消息，说没有这个名称的账号。

这就是完整的脚本。利用附录B中的指导，轻松地启动命令行程序，现在你就有了一种快速的方式，将账号的口令复制到剪贴板。如果需要更新口令，就必须修改源代码的PASSWORDS字典中的值。

当然，你可能不希望把所有的口令都放在一个地方，让某人能够轻易地复制。但你可以修改这个程序，利用它快速地将普通文本复制到剪贴板。假设你需要发出一些电子邮件，它们有许多同样的段落。你可以将每个段落作为一个值，放在PASSWORDS字典中（此时你可能希望对这个字典重命名），然后你就有了一种方式，快速地选择一些标准的文本，并复制到剪贴板。

在Windows上，你可以创建一个批处理文件，利用Win-R运行窗口，来运行这个程序（关于批处理文件的更多信息，参见附录B）。在文件编辑器中输入以下代码，保存为pw.bat，放在C:\Windows目录下：

```
@py.exe C:\Python34\pw.py %*
@pause
```

有了这个批处理文件，在Windows上运行口令保存程序，就只要按下Win-R，再输入pw <account name>。

6.4 项目：在Wiki标记中添加无序列表

在编辑一篇维基百科的文章时，你可以创建一个无序列表，即让每个列表项占据一行，并在前面放置一个星号。但是假设你有一个非常大的列表，希望添加前面的星号。你可以在每一行开始处输入这些星号，一行接一行。或者也可以用一小段Python脚本，将这个任务自动化。

bulletPointAdder.py脚本将从剪贴板中取得文本，在每一行开始处加上星号和空格，然后将这段新的文本贴回到剪贴板。例如，如果我将下面的文本复制到剪贴板（取自于维基百科的文章“List of Lists of Lists”）：

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

然后运行bulletPointAdder.py程序，剪贴板中就会包含下面的内容：

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

这段前面加了星号的文本，就可以粘贴回维基百科的文章中，成为一个无序列表。

第1步：从剪贴板中复制和粘贴

你希望bulletPointAdder.py程序完成下列事情：

1．从剪贴板粘贴文本；

2．对它做一些处理；

3．将新的文本复制到剪贴板。

第2步有一点技巧，但第1步和第3步相当简单，它们只是利用了pyperclip.copy()和pyperclip.paste()函数。现在，我们先写出程序中第1步和第3步的部分。输入以下代码，将程序保存为bulletPointAdder.py：

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()
# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

TODO注释是提醒，你最后应该完成这部分程序。下一步实际上就是实现程序的这个部分。

第2步：分离文本中的行，并添加星号

调用pyperclip.paste()将返回剪贴板上的所有文本，结果是一个大字符串。如果我们使用“List of Lists of Lists”的例子，保存在text中的字符串就像这样：

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

在打印到剪贴板，或从剪贴板粘贴时，该字符串中的\n换行字符，让它能显示为多行。在这一个字符串中有许多“行”。你想要在每一行开始处添加一个星号。

你可以编写代码，查找字符串中每个\n换行字符，然后在它后面添加一个星号。但更容易的做法是，使用split()方法得到一个字符串的列表，其中每个表项就是原来字符串中的一行，然后在列表中每个字符串前面添加星号。

让程序看起来像这样：

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list
```

```
pyperclip.copy(text)
```

我们按换行符分割文本，得到一个列表，其中每个表项是文本中的一行。我们将列表保存在lines中，然后遍历lines中的每个表项。对于每一行，我们在开始处添加一个新号和一个空格。现在lines中的每个字符串都以星号开始。

第3步：连接修改过的行

lines列表现在包含修改过的行，每行都以星号开始。但pyperclip.copy()需要一个字符串，而不是字符串的列表。要得到这个字符串，就要将lines传递给join方法，连接列表中字符串。让你的程序看起来像这样：

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)): # loop through all indexes for "lines" list

lines[i] = '* ' + lines[i] # add star to each string in "lines" list
text = '\n'.join(lines)
```

```
pyperclip.copy(text)
```

运行这个程序，它将取代剪贴板上的文本，新的文本每一行都以星号开始。现在程序完成了，可以在剪贴板中复制一些文本，试着运行它。

即使不需要自动化这样一个专门的任务，也可能想要自动化某些其他类型的文本操作，诸如删除每行末尾的空格，或将文本转换成大写或小写。不论你的需求是什么，都可以使用剪贴板作为输入和输出。

6.5 小结

文本是常见的数据形式，Python自带了许多有用的字符串方法，来处理保存在字符串中的文本。在你写的几乎每个Python程序中，都会用到取下标、切片和字符串方法。

现在你写的程序似乎不太复杂，因为它们没有图形用户界面，没有图像和彩色的文本。到目前为止，你在利用print()显示文本，利用input()让用户输入文本。但是，用户可以通过剪贴板，快速输入大量的文本。这种能力提供了一种有用的编程方式，可以操作大量的文本。这些基于文本的程序可能没有闪亮的窗口或图形，但它们能很快完成大量有用的工作。

操作大量文本的另一种方式，是直接从硬盘读写文件。在下一章中，你将学习如何用Python来做到这一点。

6.6 习题

1．什么是转义字符？

2．转义字符\n和\t代表什么？

3．如何在字符串中放入一个倒斜杠字符\？

4．字符串"Howl's Moving Castle"是有效字符串。为什么单词中的单引号没有转义，却没有问题？

5．如果你不希望在字符串中加入\n，怎样写一个带有换行的字符串？

6．下面的表达式求值为什么？

- 'Hello world!'[1]
- 'Hello world!'[0:5]
- 'Hello world!'[:5]
- 'Hello world!'[3:]

7．下面的表达式求值为什么？

- 'Hello'.upper()
- 'Hello'.upper().isupper()
- 'Hello'.upper().lower()

8．下面的表达式求值为什么？

- 'Remember, remember, the fifth of November.'.split()
- '-'.join('There can be only one.'.split())

9．什么字符串方法能用于字符串右对齐、左对齐和居中？

10．如何去掉字符串开始或末尾的空白字符？

6.7 实践项目

作为实践，编程完成下列任务。

表格打印

编写一个名为printTable()的函数，它接受字符串的列表的列表，将它显示在组织良好的表格中，每列右对齐。假定所有内层列表都包含同样数目的字符串。例如，该值可能看起来像这样：

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

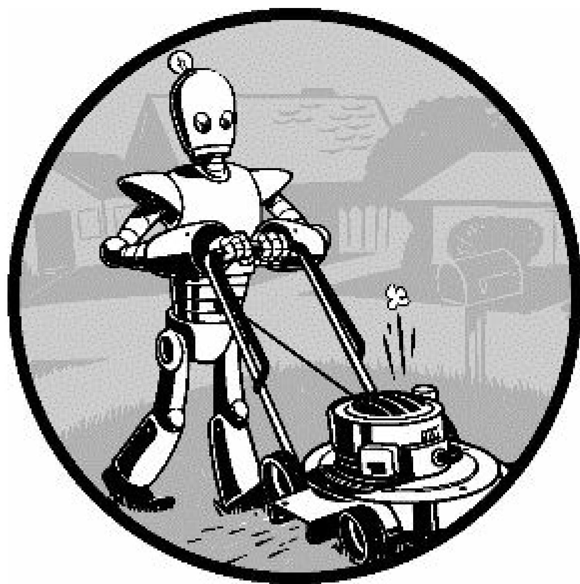
你的printTable()函数将打印出：

```
  apples Alice dogs
 oranges Bob    cats
cherries Carol moose
  banana David goose
```

提示

你的代码首先必须找到每个内层列表中最长的字符串，这样整列就有足够的宽度以放下所有字符串。你可以将每一列的最大宽度，保存为一个整数的列表。printTable()函数的开始可以是colWidths = [0] * len(tableData)，这创建了一个列表，它包含了一些0，数目与tableData中内层列表的数目相同。这样，colWidths[0]就可以保存tableData[0]中最长字符串的宽度，colWidths[1]就可以保存tableData[1]中最长字符串的宽度，以此类推。然后可以找到colWidths列表中最大的值，决定将什么整数宽度传递给rjust()字符串方法。

第二部分 自动化任务



第7章　模式匹配与正则表达式

你可能熟悉文本查找，即按下Ctrl-F，输入你要查找的词。“正则表达式”更进一步，它们让你指定要查找的“模式”。你也许不知道一家公司的准确电话号码，但如果你住在美国或加拿大，你就知道它有3位数字，然后是一个短横线，然后是4位数字（有时候以3位区号开始）。因此作为一个人，你看到一个电话号码就知道：415-555-1234是电话号码，但4,155,551,234不是。

正则表达式很有用，但如果不是程序员，很少会有人了解它，尽管大多数现代文本编辑器和文字处理器（诸如微软的Word或OpenOffice），都有查找和查找替换功能，可以根据正则表达式查找。正则表达式可以节约大量时间，不仅适用于软件用户，也适用于程序员。实际上，技术作家Cory Doctorow声称，甚至应该在教授编程之前，先教授正则表达式：

“知道[正则表达式]可能意味着用3步解决一个问题，而不是用3000步。如果你是一个技术怪侠，别忘了你用几次击键就能解决的问题，其他人需要数天的烦琐工作才能解决，而且他们容易犯错。”[1]

在本章中，你将从编写一个程序开始，先不用正则表达式来寻找文本模式。然后再看看，使用正则表达式让代码变得多么简洁。我将展示用正则表达式进行基本匹配，然后转向一些更强大的功能，诸如字符串替换，以及创建你自己的字符类型。最后，在本章末尾，你将编写一个程序，从一段文本中自动提取电话号码和E-mail地址。

7.1 不用正则表达式来查找文本模式

假设你希望在字符串中查找电话号码。你知道模式：3个数字，一个短横线，3个数字，一个短横线，再是4个数字。例如：415-555-4242。

假定我们用一个名为isPhoneNumber()的函数，来检查字符串是否匹配模式，它返回True或False。打开一个新的文件编辑器窗口，输入以下代码，然后保存为isPhoneNumber.py：

```
  def isPhoneNumber(text):
❶     if len(text) != 12:
          return False
      for i in range(0, 3):
❷         if not text[i].isdecimal():
              return False
❸     if text[3] != '-':
          return False
      for i in range(4, 7):
❹         if not text[i].isdecimal():
              return False
❺     if text[7] != '-':
          return False
      for i in range(8, 12):
❻         if not text[i].isdecimal():
              return False
❼     return True

  print('415-555-4242 is a phone number:')
  print(isPhoneNumber('415-555-4242'))
  print('Moshi moshi is a phone number:')
  print(isPhoneNumber('Moshi moshi'))
```

运行该程序，输出看起来像这样：

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

isPhoneNumber()函数的代码进行几项检查，看看text中的字符串是不是有效的电话号码。如果其中任意一项检查失败，函数就返回False。代码首先检查该字符串是否刚好有12个字符❶。然后它检查区号（就是text中的前3个字符）是否只包含数字❷。函数剩下的部分检查该字符串是否符合电话号码的模式：号码必须在区号后出现第一个短横线❸，3个数字❹，然后是另一个短横线❺，最后是4个数字❻。如果程序执行

通过了所有的检查，它就返回True❼。

用参数'415-555-4242'调用isPhoneNumber()将返回真。用参数'Moshi moshi'调用isPhoneNumber()将返回假，第一项测试失败了，因为不是12个字符。

必须添加更多代码，才能在更长的字符串中寻找这种文本模式。用下面的代码，替代isPhoneNumber.py中最后4个print()函数调用：

```
  message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
  for i in range(len(message)):
❶     chunk = message[i:i+12]
❷     if isPhoneNumber(chunk):
        print('Phone number found: ' + chunk)
  print('Done')
```

该程序运行时，输出看起来像这样：

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

在for循环的每一次迭代中，取自message的一段新的12个字符被赋给变量chunk❶。例如，在第一次迭代，i是0，chunk被赋值为message[0:12]（即字符串'Call me at 4'）。在下一次迭代，i是1，chunk被赋值为message[1:13]（字符串'all me at 41'）。

将chunk传递给isPhoneNumber()，看看它是否符合电话号码的模式❷。如果符合，就打印出这段文本。

继续遍历message，最终chunk中的12个字符会是一个电话号码。该循环遍历了整个字符串，测试了每一段12个字符，打印出所有满足isPhoneNumber()的chunk。当我们遍历完message，就打印出Done。

在这个例子中，虽然message中的字符串很短，但它也可能包含上百万个字符，程序运行仍然不需要一秒钟。使用正则表达式查找电话号码的类似程序，运行也不会超过一秒钟，但用正则表达式编写这类程序会快得多。

7.2 用正则表达式查找文本模式

前面的电话号码查找程序能工作，但它使用了很多代码，做的事却有限：isPhoneNumber()函数有17行，但只能查找一种电话号码模式。像415.555.4242或(415) 555-4242这样的电话号码格式，该怎么办呢？如果电话号码有分机，例如415-555-4242 x99，该怎么办呢？isPhoneNumber()函数在验证它们时会失败。你可以添加更多的代码来处理额外的模式，但还有更简单的方法。

正则表达式，简称为regex，是文本模式的描述方法。例如，\d 是一个正则表达式，表示一位数字字符，即任何一位 0 到 9 的数字。Python 使用正则表达式\d\d\d-\d\d\d-\d\d\d\d，来匹配前面isPhoneNumber()函数匹配的同样文本：3 个数字、一个短横线、3 个数字、一个短横线、4 个数字。所有其他字符串都不能匹配\d\d\d-\d\d\d-\d\d\d\d正则表达式。

但正则表达式可以复杂得多。例如，在一个模式后加上花括号包围的3（{3}），就是说，“匹配这个模式3次”。所以较短的正则表达式\d{3}-\d{3}-\d{4}，也匹配正确的电话号码格式。

7.2.1 创建正则表达式对象

Python中所有正则表达式的函数都在re模块中。在交互式环境中输入以下代码，导入该模块：

```
>>> import re
```

注意

本章后面的大多数例子都需要re模块，所以要记得在你写的每个脚本开始处导入它，或重新启动IDLE时。否则，就会遇到错误消息NameError: name 're' is not defined。

向re.compile()传入一个字符串值，表示正则表达式，它将返回一个Regex模式对象（或者就简称为Regex对象）。

要创建一个Regex对象来匹配电话号码模式，就在交互式环境中输入以下代码（回忆一下，\d表示“一个数字字符”，\d\d\d-\d\d\d-\d\d\d\d是正确电话号码模式的正则表达式）。

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

现在phoneNumRegex变量包含了一个Regex对象。

7.2.2 匹配Regex对象

Regex对象的search()方法查找传入的字符串，寻找该正则表达式的所有匹配。如果字符串中没有找到该正则表达式模式，search()方法将返回None。如果找到了该模式，search()方法将返回一个Match对象。Match对象有一个group()方法，它返回被查找字符串中实际匹配的文本（稍后我会解释分组）。例如，在交互式环境中输入以下代码：

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

```
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')

>>> print('Phone number found: ' + mo.group())

Phone number found: 415-555-4242
```

变量名mo是一个通用的名称，用于Match对象。这个例子可能初看起来有点复杂，但它比前面的isPhoneNumber.py程序要短很多，并且做的事情一样。

这里，我们将期待的模式传递给re.compile()，并将得到的Regex对象保存在phoneNumRegex中。然后我们在phoneNumRegex上调用search()，向它传入想查找的字符串。查找的结果保存在变量mo中。在这个例子里，我们知道模式会在这个字符串中找到，所以我们知道会返回一个Match对象。知道mo包含一个Match对象，而不是空值None，我们就可以在mo变量上调用group()，返回匹配的结果。将mo.group()写在打印语句中，显示出完整的匹配，即415-555-4242。

向re.compile()传递原始字符串

回忆一下，Python中转义字符使用倒斜杠（\）。字符串'\n'表示一个换行字符，而不是倒斜杠加上一个小写的n。你需要输入转义字符\，才能打印出一个倒斜杠。所以'\n'表示一个倒斜杠加上一个小写的n。但是，通过在字符串的第一个引号之前加上r，可以将该字符串标记为原始字符串，它不包括转义字符。

因为正则表达式常常使用倒斜杠，向re.compile()函数传入原始字符串就很方便，而不是输入额外得到斜杠。输入r'\d\d\d-\d\d\d-\d\d\d\d'，比输入'\d\d\d-\d\d\d-\d\d\d\d'要容易得多。

7.2.3 正则表达式匹配复习

虽然在Python中使用正则表达式有几个步骤，但每一步都相当简单。

1．用import re导入正则表达式模块。

2．用re.compile()函数创建一个Regex对象（记得使用原始字符串）。

3．向Regex对象的search()方法传入想查找的字符串。它返回一个Match对象。

4．调用Match对象的group()方法，返回实际匹配文本的字符串。

> 注意
>
> 虽然我鼓励你在交互式环境中输入示例代码,但你也应该利用基于网页的正则表达式测试程序。它可以向你清楚地展示，一个正则表达式如何匹配输入的一段文本。我推荐的测试程序位于http://regexpal.com/ 。

7.3 用正则表达式匹配更多模式

既然你已知道用Python创建和查找正则表达式对象的基本步骤，就可以尝试一些更强大的模式匹配功能了。

7.3.1 利用括号分组

假定想要将区号从电话号码中分离。添加括号将在正则表达式中创建“分组”：(\d\d\d)-(\d\d\d-\d\d\d\d)。然后可以使用group()匹配对象方法，从一个分组中获取匹配的文本。

正则表达式字符串中的第一对括号是第1组。第二对括号是第2组。向group()匹配对象方法传入整数1或2，就可以取得匹配文本的不同部分。向group()方法传入0或不传入参数，将返回整个匹配的文本。在交互式环境中输入以下代码：

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
```

```
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

如果想要一次就获取所有的分组，请使用groups()方法，注意函数名的复数形式。

```
>>> mo.groups()
```

```
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

因为mo.groups()返回多个值的元组，所以你可以使用多重复制的技巧，每个值赋给一个独立的变量，就像前面的代码行：areaCode, mainNumber = mo.groups()。

括号在正则表达式中有特殊的含义，但是如果你需要在文本中匹配括号，怎么办？例如，你要匹配的电话号码，可能将区号放在一对括号中。在这种情况下，就需要用倒斜杠对(和)进行字符转义。在交互式环境中输入以下代码：

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
```

```
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

传递给re.compile()的原始字符串中，(和)转义字符将匹配实际的括号字符。

7.3.2 用管道匹配多个分组

字符|称为“管道”。希望匹配许多表达式中的一个时，就可以使用它。例如，正则表达式r'Batman|Tina Fey'将匹配'Batman'或'Tina Fey'。

如果Batman和Tina Fey都出现在被查找的字符串中，第一次出现的匹配文本，将作为Match对象返回。在交互式环境中输入以下代码：

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
```

```
>>> mo1 = heroRegex.search('Batman and Tina Fey.')

>>> mo1.group()

'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')

>>> mo2.group()

'Tina Fey'
```

注意

利用findall()方法，可以找到“所有”匹配的地方。这在7.5节“findall()方法”中讨论。

也可以使用管道来匹配多个模式中的一个，作为正则表达式的一部分。例如，假设你希望匹配'Batman'、'Batmobile'、'Batcopter'和'Batbat'中任意一个。因为所有这些字符串都以Bat开始，所以如果能够只指定一次前缀，就很方便。这可以通过括号实现。在交互式环境中输入以下代码：

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
```

```
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

方法调用mo.group()返回了完全匹配的文本'Batmobile'，而mo.group(1)只是返回第一个括号分组内匹配的文本'mobile'。通过使用管道字符和分组括号，可以指定几种可选的模式，让正则表达式去匹配。

如果需要匹配真正的管道字符，就用倒斜杠转义，即|。

7.3.3 用问号实现可选匹配

有时候，想匹配的模式是可选的。就是说，不论这段文本在不在，正则表达式都会认为匹配。字符?表明它前面的分组在这个模式中是可选的。例如，在交互式环境中输入以下代码：

```
>>> batRegex = re.compile(r'Bat(wo)?man')
```

```
>>> mo1 = batRegex.search('The Adventures of Batman')

>>> mo1.group()

'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')

>>> mo2.group()

'Batwoman'
```

正则表达式中的(wo)?部分表明，模式wo是可选的分组。该正则表达式匹配的文本中，wo将出现零次或一次。这就是为什么正则表达式既匹配'Batwoman'，又匹配'Batman'。

利用前面电话号码的例子，你可以让正则表达式寻找包含区号或不包含区号的电话号码。在交互式环境中输入以下代码：

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
```

```
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'
>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

你可以认为?是在说，“匹配这个问号之前的分组零次或一次”。

如果需要匹配真正的问号字符，就使用转义字符\?。

7.3.4 用星号匹配零次或多次

*（称为星号）意味着“匹配零次或多次”，即星号之前的分组，可以在文本中出现任意次。它可以完全不存在，或一次又一次地重复。让

我们再来看看Batman的例子。

```
>>> batRegex = re.compile(r'Bat(wo)*man')

>>> mo1 = batRegex.search('The Adventures of Batman')

>>> mo1.group()

'Batman'
>>> mo2 = batRegex.search('The Adventures of Batwoman')

>>> mo2.group()

'Batwoman'
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')

>>> mo3.group()
```

```
'Batwowowowoman'
```

对于'Batman'，正则表达式的(wo)部分匹配*wo*的零个实例。对于*'Batwoman'*，*(wo)* 匹配wo的一个实例。对于'Batwowowowoman'，(wo)*匹配wo的4个实例。

如果需要匹配真正的星号字符，就在正则表达式的星号字符前加上倒斜杠，即*。

7.3.5 用加号匹配一次或多次

*意味着“匹配零次或多次”，+（加号）则意味着“匹配一次或多次”。星号不要求分组出现在匹配的字符串中，但加号不同，加号前面的分组必须“至少出现一次”。这不是可选的。在交互式环境中输入以下代码，把它和前一节的星号正则表达式进行比较：

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
```

```
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

正则表达式Bat(wo)+man不会匹配字符串'The Adventures of Batman'，因为加号要求wo至少出现一次。

如果需要匹配真正的加号字符，在加号前面加上倒斜杠实现转义：+。

7.3.6 用花括号匹配特定次数

如果想要一个分组重复特定次数，就在正则表达式中该分组的后面，跟上花括号包围的数字。例如，正则表达式(Ha){3}将匹配字符串'HaHaHa'，但不会匹配'HaHa'，因为后者只重复了(Ha)分组两次。

除了一个数字，还可以指定一个范围，即在花括号中写下一个最小值、一个逗号和一个最大值。例如，正则表达式(Ha){3,5}将匹配'HaHaHa'、'HaHaHaHa'和'HaHaHaHaHa'。

也可以不写花括号中的第一个或第二个数字，不限定最小值或最大值。例如，(Ha){3,}将匹配3次或更多次实例，(Ha){,5}将匹配0到5次实例。花括号让正则表达式更简短。这两个正则表达式匹配同样的模式：

```
(Ha){3}
(Ha)(Ha)(Ha)
```

这两个正则表达式也匹配同样的模式：

```
(Ha){3,5}
((Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha))|((Ha)(Ha)(Ha)(Ha)(Ha))
```

在交互式环境中输入以下代码：

```
>>> haRegex = re.compile(r'(Ha){3}')

>>> mo1 = haRegex.search('HaHaHa')

>>> mo1.group()
```

```
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

这里，(Ha){3}匹配'HaHaHa'，但不匹配'Ha'。因为它不匹配'Ha'，所以search()返回None。

7.4 贪心和非贪心匹配

在字符串'HaHaHaHaHa'中，因为(Ha){3,5}可以匹配3个、4个或5个实例，你可能会想，为什么在前面花括号的例子中，Match对象的group()调用会返回'HaHaHaHaHa'，而不是更短的可能结果。毕竟，'HaHaHa'和'HaHaHaHa'也能够有效地匹配正则表达式(Ha){3,5}。

Python的正则表达式默认是“贪心”的，这表示在有二义的情况下，它们会尽可能匹配最长的字符串。花括号的“非贪心”版本匹配尽可能最短的字符串，即在结束的花括号后跟着一个问号。

在交互式环境中输入以下代码，注意在查找相同字符串时，花括号的贪心形式和非贪心形式之间的区别：

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
```

```
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')

>>> mo1.group()

'HaHaHaHaHa'
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')

>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')

>>> mo2.group()

'HaHaHa'
```

请注意，问号在正则表达式中可能有两种含义：声明非贪心匹配或表示可选的分组。这两种含义是完全无关的。

7.5 findall()方法

除了search方法外，Regex对象也有一个findall()方法。search()将返回一个Match对象，包含被查找字符串中的“第一次”匹配的文本，而findall()方法将返回一组字符串，包含被查找字符串中的所有匹配。为了看看search()返回的Match对象只包含第一次出现的匹配文本，请在交互式环境中输入以下代码：

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')

>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work: 212-555-0000')

>>> mo.group()

'415-555-9999'
```

另一方面，findall()不是返回一个Match对象，而是返回一个字符串列表，只要在正则表达式中没有分组。列表中的每个字符串都是一段被查找的文本，它匹配该正则表达式。在交互式环境中输入以下代码：

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
```

```
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')

['415-555-9999', '212-555-0000']
```

如果在正则表达式中有分组，那么findall将返回元组的列表。每个元组表示一个找到的匹配，其中的项就是正则表达式中每个分组的匹配字符串。为了看看findall()的效果，请在交互式环境中输入以下代码（请注意，被编译的正则表达式现在有括号分组）：

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has group

>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')

[('415', '555', '1122'), ('212', '555', '0000')]
```

作为findall()方法的返回结果的总结，请记住下面两点：

1．如果调用在一个没有分组的正则表达式上，例如\d\d\d-\d\d\d-\d\d\d\d，方法findall()将返回一个匹配字符串的列表，例如['415-555-9999', '212-555-0000']。

2．如果调用在一个有分组的正则表达式上，例如(\d\d\d)-(\d\d\d)-

(\d\d\d\d)，方法findall()将返回一个字符串的元组的列表（每个分组对应一个字符串），例如[('415', '555', '1122'), ('212', '555', '0000')]。

7.6 字符分类

在前面电话号码正则表达式的例子中，你知道\d 可以代表任何数字。也就是说，\d是正则表达式(0|1|2|3|4|5|6|7|8|9)的缩写。有许多这样的“缩写字符分类”，如表7-1所示。

表**7-1** 常用字符分类的缩写代码

缩写字符分类	表示
\d	0到9的任何数字
\D	除0到9的数字以外的任何字符
\w	任何字母、数字或下划线字符（可以认为是匹配“单词”字符）
\W	除字母、数字和下划线以外的任何字符
\s	空格、制表符或换行符（可以认为是匹配“空白”字符）
\S	除空格、制表符和换行符以外的任何字符

字符分类对于缩短正则表达式很有用。字符分类[0-5]只匹配数字0到5，这比输入(0|1|2|3|4|5)要短很多。

例如，在交互式环境中输入以下代码：

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
```

```
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids,

swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')

['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans',
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

正则表达式\d+\s\w+匹配的文本有一个或多个数字(\d+)，接下来是一个空白字符(\s)，接下来是一个或多个字母/数字/下划线字符(\w+)。findall()方法将返回所有匹配该正则表达式的字符串，放在一个列表中。

7.7 建立自己的字符分类

有时候你想匹配一组字符，但缩写的字符分类（\d、\w、\s等）太宽泛。你可以用方括号定义自己的字符分类。例如，字符分类[aeiouAEIOU]将匹配所有元音字符，不论大小写。在交互式环境中输入以下代码：

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')

>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')
```

```
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

也可以使用短横表示字母或数字的范围。例如，字符分类[a-zA-Z0-9]将匹配所有小写字母、大写字母和数字。

请注意，在方括号内，普通的正则表达式符号不会被解释。这意味着，你不需要前面加上倒斜杠转义.、*、?或()字符。例如，字符分类将匹配数字0到5和一个句点。你不需要将它写成[0-5.]。

通过在字符分类的左方括号后加上一个插入字符（^），就可以得到“非字符类”。非字符类将匹配不在这个字符类中的所有字符。例如，在交互式环境中输入以下代码：

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')

>>> consonantRegex.findall('RoboCop eats baby food. BABY FOOD.')

['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.',
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

现在，不是匹配所有元音字符，而是匹配所有非元音字符。

7.8 插入字符和美元字符

可以在正则表达式的开始处使用插入符号（^），表明匹配必须发生在被查找文本开始处。类似地，可以再正则表达式的末尾加上美元符号（$），表示该字符串必须以这个正则表达式的模式结束。可以同时使用^和$，表明整个字符串必须匹配该模式，也就是说，只匹配该字符串的某个子集是不够的。

例如，正则表达式r'^Hello'匹配以'Hello'开始的字符串。在交互式环境中输入以下代码：

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
< _sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

正则表达式r'\d$'匹配以数字0到9结束的字符串。在交互式环境中输入以下代码：

```
>>> endsWithNumber = re.compile(r'\d$')
```

```
>>> endsWithNumber.search('Your number is 42')

< _sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None

True
```

正则表达式r'^\d+$'匹配从开始到结束都是数字的字符串。在交互式环境中输入以下代码：

```
>>> wholeStringIsNum = re.compile(r'^\d+$')

>>> wholeStringIsNum.search('1234567890')

< _sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None

True
```

```
>>> wholeStringIsNum.search('12 34567890') == None
True
```

前面交互式脚本例子中的最后两次search()调用表明，如果使用了^和$，那么整个字符串必须匹配该正则表达式。

我总是会混淆这两个符号的含义，所以我使用助记法“Carrots cost dollars”，提醒我插入符号在前面，美元符号在后面。

7.9 通配字符

在正则表达式中，.（句点）字符称为“通配符”。它匹配除了换行之外的所有字符。例如，在交互式环境中输入以下代码：

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

要记住，句点字符只匹配一个字符，这就是为什么在前面的例子中，对于文本flat，只匹配lat。要匹配真正的句点，就是用倒斜杠转义：.。

7.9.1 用点-星匹配所有字符

有时候想要匹配所有字符串。例如，假定想要匹配字符串'First Name:'，接下来是任意文本，接下来是'Last Name:'，然后又是任意文本。可以用点-星（.*）表示“任意文本”。回忆一下，句点字符表示“除换行外所有单个字符”，星号字符表示“前面字符出现零次或多次”。

在交互式环境中输入以下代码：

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

点-星使用“贪心”模式：它总是匹配尽可能多的文本。要用“非贪心”模式匹配所有文本，就使用点-星和问号。像和大括号一起使用时那样，问号告诉Python用非贪心模式匹配。在交互式环境中输入以下代码，看看贪心模式和非贪心模式的区别：

```
>>> nongreedyRegex = re.compile(r'<.*?>')

>>> mo = nongreedyRegex.search(' for dinner.>')

>>> mo.group()

'< To serve man>'

>>> greedyRegex = re.compile(r'<.*>')

>>> mo = greedyRegex.search(' for dinner.>')

>>> mo.group()

'< To serve man> for dinner.>'
```

两个正则表达式都可以翻译成“匹配一个左尖括号，接下来是任意字符，接下来是一个右尖括号”。但是字符串'<To serve man> for dinner.>'对右肩括号有两种可能的匹配。在非贪心的正则表达式中，Python匹配最短可能的字符串：'<To serve man>'。在贪心版本中，Python匹配最长可能的字符串：'<To serve man> for dinner.>'。

7.9.2 用句点字符匹配换行

点-星将匹配除换行外的所有字符。通过传入re.DOTALL作为re.compile()的第二个参数，可以让句点字符匹配所有字符，包括换行字符。

在交互式环境中输入以下代码：

```
>>> noNewlineRegex = re.compile('.*')

>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.

\nUphold the law.').group()

'Serve the public trust.'
>>> newlineRegex = re.compile('.*', re.DOTALL)
```

```
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

正则表达式noNewlineRegex在创建时没有向re.compile()传入re.DOTALL，它将匹配所有字符，直到第一个换行字符。但是，newlineRegex在创建时向re.compile()传入了re.DOTALL，它将匹配所有字符。这就是为什么newlineRegex.search()调用匹配完整的字符串，包括其中的换行字符。

7.10 正则表达式符号复习

本章介绍了许多表示法，所以这里快速复习一下学到的内容：

- ?匹配零次或一次前面的分组。
- *匹配零次或多次前面的分组。
- +匹配一次或多次前面的分组。
- {n}匹配n次前面的分组。
- {n,}匹配n次或更多前面的分组。
- {,m}匹配零次到m次前面的分组。
- {n,m}匹配至少n次、至多m次前面的分组。
- {n,m}?或*?或+?对前面的分组进行非贪心匹配。
- ^spam意味着字符串必须以spam开始。
- spam$意味着字符串必须以spam结束。
- .匹配所有字符，换行符除外。

- \d、\w和\s分别匹配数字、单词和空格。
- \D、\W和\S分别匹配出数字、单词和空格外的所有字符。
- [abc]匹配方括号内的任意字符（诸如a、b或c）。
- [^abc]匹配不在方括号内的任意字符。

7.11 不区分大小写的匹配

通常，正则表达式用你指定的大小写匹配文本。例如，下面的正则表达式匹配完全不同的字符串：

```
>>> regex1 = re.compile('RoboCop')




>>> regex2 = re.compile('ROBOCOP')




>>> regex3 = re.compile('robOcop')




>>> regex4 = re.compile('RobocOp')
```

但是，有时候你只关心匹配字母，不关心它们是大写或小写。要让正则表达式不区分大小写，可以向re.compile()传入re.IGNORECASE或re.I，作为第二个参数。在交互式环境中输入以下代码：

```
>>> robocop = re.compile(r'robocop', re.I)

>>> robocop.search('RoboCop is part man, part machine, all cop.').group()

'RoboCop'

>>> robocop.search('ROBOCOP protects the innocent.').group()

'ROBOCOP'

>>> robocop.search('Al, why does your programming book talk about robocop s

'robocop'
```

7.12 用**sub()**方法替换字符串

正则表达式不仅能找到文本模式，而且能够用新的文本替换掉这些模式。Regex对象的sub()方法需要传入两个参数。第一个参数是一个字符串，用于取代发现的匹配。第二个参数是一个字符串，即正则表达式。sub()方法返回替换完成后的字符串。

例如，在交互式环境中输入以下代码：

```
>>> namesRegex = re.compile(r'Agent \w+')

>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Ag

'CENSORED gave the secret documents to CENSORED.'
```

有时候，你可能需要使用匹配的文本本身，作为替换的一部分。在sub()的第一个参数中，可以输入\1、\2、\3……。表示"在替换中输入分组1、2、3……的文本"。

例如，假定想要隐去密探的姓名，只显示他们姓名的第一个字母。要做到这一点，可以使用正则表达式Agent (\w)\w，传入*r'\1* *'作为sub()的第一个参数。字符串中的\1将由分组1匹配的文本所替代，也就是正则表达式的(\w)分组。

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')

>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent

Eve knew Agent Bob was a double agent.')
```

```
A**** told C**** that E**** knew B**** was a double agent.'
```

7.13 管理复杂的正则表达式

如果要匹配的文本模式很简单，正则表达式就很好。但匹配复杂的文本模式，可能需要长的、费解的正则表达式。你可以告诉re.compile()，忽略正则表达式字符串中的空白符和注释，从而缓解这一点。要实现这种详细模式，可以向re.compile()传入变量re.VERBOSE，作为第二个参数。

现在，不必使用这样难以阅读的正则表达式：

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

你可以将正则表达式放在多行中，并加上注释，像这样：

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

请注意，前面的例子使用了三重引号(''')，创建了一个多行字符

串。这样就可以将正则表达式定义放在多行中，让它更可读。

正则表达式字符串中的注释规则，与普通的Python代码一样：#符号和它后面直到行末的内容，都被忽略。而且，表示正则表达式的多行字符串中，多余的空白字符也不认为是要匹配的文本模式的一部分。这让你能够组织正则表达式，让它更可读。

7.14 组合使用re.IGNOREC ASE、re.DOTALL和re.VERBOSE

如果你希望在正则表达式中使用re.VERBOSE来编写注释，还希望使用re.IGNORECASE来忽略大小写，该怎么办？遗憾的是，re.compile()函数只接受一个值作为它的第二参数。可以使用管道字符（|）将变量组合起来，从而绕过这个限制。管道字符在这里称为“按位或”操作符。

所以，如果希望正则表达式不区分大小写，并且句点字符匹配换行，就可以这样构造re.compile()调用：

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

使用第二个参数的全部3个选项，看起来像这样：

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBO
```

这个语法有一点老式，源自于早期的Python版本。位运算符的细节超出了本书的范围，更多的信息请查看资源http://nostarch.com/automatestuff/ 。可以向第二个参数传入其他选项，它们不常用，但你也可以在前面的资源中找到有关它们的信息。

7.15 项目：电话号码和E-mail地址提取程序

假设你有一个无聊的任务，要在一篇长的网页或文章中，找出所有电话号码和邮件地址。如果手动翻页，可能需要查找很长时间。如果有一个程序，可以在剪贴板的文本中查找电话号码和E-mail地址，那你就只要按一下Ctrl-A选择所有文本，按下Ctrl-C将它复制到剪贴板，然后运行你的程序。它会用找到的电话号码和E-mail地址，替换掉剪贴板中的文本。

当你开始接手一个新项目时，很容易想要直接开始写代码。但更多的时候，最好是后退一步，考虑更大的图景。我建议先草拟高层次的计划，弄清楚程序需要做什么。暂时不要思考真正的代码，稍后再来考虑。现在，先关注大框架。

例如，你的电话号码和E-mail地址提取程序需要完成以下任务：

- 从剪贴板取得文本。
- 找出文本中所有的电话号码和E-mail地址。
- 将它们粘贴到剪贴板。

现在你可以开始思考，如何用代码来完成工作。代码需要做下面的事情：

- 使用pyperclip模块复制和粘贴字符串。
- 创建两个正则表达式，一个匹配电话号码，另一个匹配E-mail地址。
- 对两个正则表达式，找到所有的匹配，而不只是第一次匹配。
- 将匹配的字符串整理好格式，放在一个字符串中，用于粘贴。

- 如果文本中没有找到匹配，显示某种消息。

这个列表就像项目的路线图。在编写代码时，可以独立地关注其中的每一步。每一步都很好管理。它的表达方式让你知道在Python中如何去做。

第**1**步：为电话号码创建一个正则表达式

首先，你需要创建一个正则表达式来查找电话号码。创建一个新文件，输入以下代码，保存为phoneAndEmail.py：

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboa

import pyperclip, re

phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    (\d{3})                       # first 3 digits
    (\s|-|\.)                     # separator
    (\d{4})                       # last 4 digits
    (\s*(ext|x|ext.)\s*(\d{2,5}))?     # extension
    )''', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

TODO注释仅仅是程序的框架。当编写真正的代码时，它们会被替换掉。

电话号码从一个“可选的”区号开始，所以区号分组跟着一个问号。因为区号可能只是3个数字（即\d{3}），或括号中的3个数字（即(\d{3})），所以应该用管道符号连接这两部分。可以对这部分多行字符串加上正则表达式注释# Area code，帮助你记忆(\d{3}|(\d{3}))?要匹配

的是什么。

电话号码分割字符可以是空格（\s）、短横（-）或句点（.），所以这些部分也应该用管道连接。这个正则表达式接下来的几部分很简单：3个数字，接下来是另一个分割符，接下来是4个数字。最后的部分是可选的分机号，包括任意数目的空格，接着ext、x或ext.，再接着2到5位数字。

第2步：为E-mail地址创建一个正则表达式

还需要一个正则表达式来匹配E-mail地址。让你的程序看起来像这样：

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipb
import pyperclip, re

phoneRegex = re.compile(r'''(
--_snip_--

# Create email regex.

emailRegex = re.compile(r'''(

❶    [a-zA-Z0-9._%+-]+         # username

❷    @                         # @ symbol
```

```
❸     [a-zA-Z0-9.-]+         # domain name

      (\.[a-zA-Z]{2,4})     # dot-something

      )''', re.VERBOSE)

  # TODO: Find matches in clipboard text.

  # TODO: Copy results to the clipboard.
```

E-mail地址的用户名部分❶是一个或多个字符，字符可以包括：小写和大写字母、数字、句点、下划线、百分号、加号或短横。可以将所有这些放入一个字符分类：[a-zA-Z0-9._%+-]。

域名和用户名用@符号分割❷，域名❸允许的字符分类要少一些，只允许字母、数字、句点和短横：[a-zA-Z0-9.-]。最后是“dot-com”部分（技术上称为“顶级域名”），它实际上可以是“dot-anything”。它有2到4个字符。

E-mail地址的格式有许多奇怪的规则。这个正则表达式不会匹配所有可能的、有效的E-mail地址，但它会匹配你遇到的大多数典型的电子

邮件地址。

第3步：在剪贴板文本中找到所有匹配

既然已经指定了电话号码和电子邮件地址的正则表达式，就可以让Python的re模块做辛苦的工作，查找剪贴板文本中所有的匹配。pyperclip.paste()函数将取得一个字符串，内容是剪贴板上的文本，findall()正则表达式方法将返回一个元组的列表。

让你的程序看起来像这样：

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipb

import pyperclip, re

phoneRegex = re.compile(r′ ′ ′ (
--_snip_--

# Find matches in clipboard text.

text = str(pyperclip.paste())

❶ matches = []

❷ for groups in phoneRegex.findall(text):

    phoneNum = '-'.join([groups[1], groups[3], groups[5]])
```

```
        if groups[8] != '':
            phoneNum += ' x' + groups[8]
        matches.append(phoneNum)
❸ for groups in emailRegex.findall(text):
        matches.append(groups[0])

   # TODO: Copy results to the clipboard.
```

每个匹配对应一个元组，每个元组包含正则表达式中每个分组的字符串。回忆一下，分组0匹配整个正则表达式，所以在元组下标0处的分组，就是你感兴趣的内容。

在❶处可以看到，你将所有的匹配保存在名为matches的列表变量中。它从一个空列表开始，经过几个for循环。对于E-mail地址，你将每次匹配的分组0添加到列表中❸。对于匹配的电话号码，你不想只是添加分组0。虽然程序可以“检测”几种不同形式的电话号码，你希望添加的电话号码是唯一的、标准的格式。phoneNum变量包含一个字符串，它由匹配文本的分组1、3、5和8构成❷。（这些分组是区号、前3个数字、后4个数字和分机号。）

第4步：所有匹配连接成一个字符串，复制到剪贴板

现在，E-mail地址和电话号码已经作为字符串列表放在matches中，你希望将它们复制到剪贴板。pyperclip.copy()函数只接收一个字符串值，而不是字符串的列表，所以你在matches上调用join()方法。

为了更容易看到程序在工作，让我们将所有找到的匹配都输出在终端上。如果没有找到电话号码或E-mail地址，程序应该告诉用户。

让你的程序看起来像这样：

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboa

--_snip_--
for groups in emailRegex.findall(text):







matches.append(groups[0])

# Copy results to the clipboard.
```

```
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

第5步：运行程序

作为一个例子，打开你的Web浏览器，访问No Starch Press的联系页面http://www.nostarch.com/contactus.htm。按下Ctrl-A选择该页的所有

文本，按下Ctrl-C将它复制到剪贴板。运行这个程序，输出看起来像这样：

```
Copied to clipboard:
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

第6步：类似程序的构想

识别文本的模式（并且可能用sub()方法替换它们）有许多不同潜在的应用。

- 寻找网站的URL，它们以http://或https://开始。
- 整理不同日期格式的日期（诸如3/14/2015、03-14-2015和2015/3/14），用唯一的标准格式替代。
- 删除敏感的信息，诸如社会保险号或信用卡号。
- 寻找常见打字错误，诸如单词间的多个空格、不小心重复的单词，或者句子末尾处多个感叹号。它们很烦人！！

7.16 小结

虽然计算机可以很快地查找文本，但你必须精确地告诉它要找什么。正则表达式让你精确地指明要找的文本模式。实际上，某些文字处理和电子表格应用提供了查找替换功能，让你使用正则表达式进行查找。

Python自带的re模块让你编译Regex对象。该对象有几种方法：search()查找单词匹配，findall()查找所有匹配实例，sub()对文本进行查找和替换。

除本章介绍的语法以外，还有一些正则表达式语法。你可以在官方Python文档中找到更多内容：http://docs.python.org/3/library/re.html 。指南网站http://www.regular- expressions.info/ 也是很有用的资源。

既然已经掌握了如何操纵和匹配字符串，接下来就该学习如何在计算机硬盘上读写文件了。

7.17 习题

1．创建Regex对象的函数是什么？

2．在创建Regex对象时，为什么常用原始字符串？

3．search()方法返回什么？

4．通过Match对象，如何得到匹配该模式的实际字符串？

5．用r'(\d\d\d)-(\d\d\d-\d\d\d\d)'创建的正则表达式中，分组0表示什么？分组1呢？分组2呢？

6．括号和句点在正则表达式语法中有特殊的含义。如何指定正则表达式匹配真正的括号和句点字符？

7．findall()方法返回一个字符串的列表，或字符串元组的列表。是什么决定它提供哪种返回？

8．在正则表达式中，|字符表示什么意思？

9．在正则表达式中，?字符有哪两种含义？

10．在正则表达式中，+和*字符之间的区别是什么？

11．在正则表达式中，{3}和{3,5}之间的区别是什么？

12．在正则表达式中，\d、\w和\s缩写字符类是什么意思？

13．在正则表达式中，\D、\W和\S缩写字符类是什么意思？

14．如何让正则表达式不区分大小写？

15．字符.通常匹配什么？如果re.DOTALL作为第二个参数传递给re.compile()，它会匹配什么？

16．.和 ?之间的区别是什么？

17．匹配所有数字和小写字母的字符分类语法是什么？

18．如果numRegex = re.compile(r'\d+')，那么numRegex.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')返回什么？

19．将re.VERBOSE作为第二个参数传递给re.compile()，让你能做什么？

20．如何写一个正则表达式，匹配每3位就有一个逗号的数字？它必须匹配以下数字：

- '42'
- '1,234'
- '6,368,745'

但不会匹配：

- '12,34,567' （逗号之间只有两位数字）
- '1234' （缺少逗号）

21．如何写一个正则表达式，匹配姓Nakamoto的完整姓名？你可以假定名字总是出现在姓前面，是一个大写字母开头的单词。该正则表达式必须匹配：

- 'Satoshi Nakamoto'
- 'Alice Nakamoto'
- 'RoboCop Nakamoto'

但不匹配：

- 'satoshi Nakamoto'（名字没有大写首字母）
- 'Mr. Nakamoto'（前面的单词包含非字母字符）

- 'Nakamoto' （没有名字）
- 'Satoshi nakamoto'（姓没有首字母大写）

22．如何编写一个正则表达式匹配一个句子，它的第一个词是Alice、Bob或Carol，第二个词是eats、pets或throws，第三个词是apples、cats或baseballs。该句子以句点结束。这个正则表达式应该不区分大小写。它必须匹配：

- 'Alice eats apples.'
- 'Bob pets cats.'
- 'Carol throws baseballs.'
- 'Alice throws Apples.'
- 'BOB EATS CATS.'

但不匹配：

- 'RoboCop eats apples.'
- 'ALICE THROWS FOOTBALLS.'
- 'Carol eats 7 cats.'

7.18 实践项目

作为实践，编程完成下列任务。

7.18.1 强口令检测

写一个函数，它使用正则表达式，确保传入的口令字符串是强口令。强口令的定义是：长度不少于8个字符，同时包含大写和小写字符，至少有一位数字。你可能需要用多个正则表达式来测试该字符串，以保证它的强度。

7.18.2 strip()的正则表达式版本

写一个函数，它接受一个字符串，做的事情和strip()字符串方法一样。如果只传入了要去除的字符串，没有其他参数，那么就从该字符串首尾去除空白字符。否则，函数第二个参数指定的字符将从该字符串中去除。

---

[1] Cory Doctorow, “Here’s what ICT should really teach kids: how to do regular expressions,”_Guardian_, December 4, 2012, _http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/._

第8章　读写文件

当程序运行时，变量是保存数据的好方法，但如果希望程序结束后数据仍然保持，就需要将数据保存到文件中。你可以认为文件的内容是一个字符串值，大小可能有几个GB。在本章中，你将学习如何使用Python在硬盘上创建、读取和保存文件。

8.1 文件与文件路径

文件有两个关键属性："文件名"（通常写成一个单词）和"路径"。路径指明了文件在计算机上的位置。例如，我的 Windows 7 笔记本上有一个文件名为projects.docx，它的路径在C:\Users\asweigart\Documents。文件名中，最后一个句点之后的部分称为文件的"扩展名"，它指出了文件的类型。project.docx是一个Word文档，Users、asweigart和Documents都是指"文件夹"（也成为目录）。文件夹可以包含文件和其他文件夹。例如，project.docx在Documents文件夹中，该文件夹又在asweigart文件夹中，asweigart文件夹又在Users文件夹中。图8-1展示了这个文件夹的组织结构。

路径中的C:\部分是"根文件夹"，它包含了所有其他文件夹。在Windows中，根文件夹名为C:\，也称为C：盘。在OS X和Linux中，根文件夹是/。在本书中，我使用Windows风格的根文件夹，C:\。如果你在OS X或Linux上输入交互式环境的例子，请用/代替。

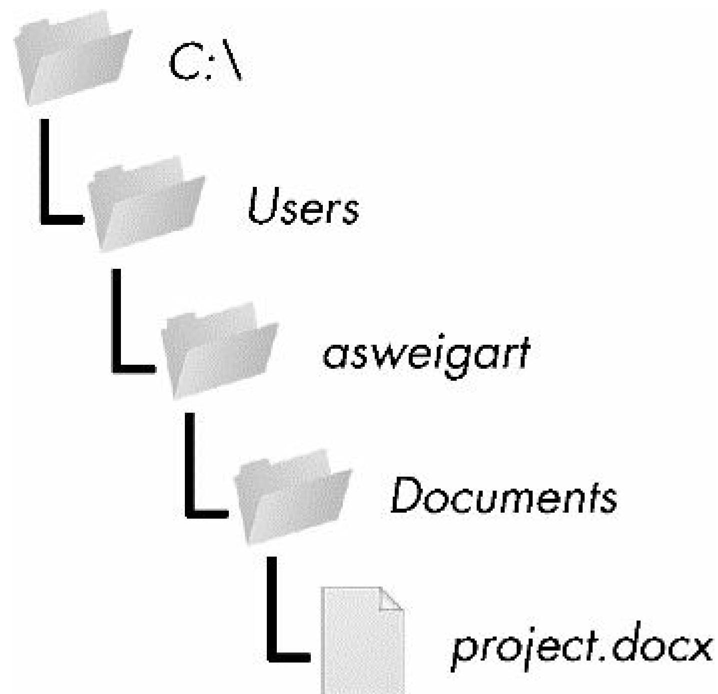


图8-1 在文件夹层次结构中的一个文件

附加卷，诸如DVD驱动器或USB闪存驱动器，在不同的操作系统上显示也不同。在Windows上，它们表示为新的、带字符的根驱动器。诸如D:\或E:\。在OS X上，它们表示为新的文件夹，在/Volumes文件夹下。在Linux上，它们表示为新的文件夹，在/mnt（"mount"）文件夹下。同时也要注意，虽然文件夹名称和文件名在Windows和OS X上是不区分大小写的，但在Linux上是区分大小写的。

8.1.1 Windows上的倒斜杠以及OS X和Linux上的正斜杠

在Windows上，路径书写使用倒斜杠作为文件夹之间的分隔符。但在OS X和Linux上，使用正斜杠作为它们的路径分隔符。如果想要程序运行在所有操作系统上，在编写Python脚本时，就必须处理这两种情况。

好在，用os.path.join()函数来做这件事很简单。如果将单个文件和路径上的文件夹名称的字符串传递给它，os.path.join()就会返回一个文件路径的字符串，包含正确的路径分隔符。在交互式环境中输入以下代码：

```
>>> import os
```

```
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

我在Windows上运行这些交互式环境的例子，所以，os.path
.join('usr', 'bin', 'spam')返回'usr\bin\spam'（请注意，倒斜杠有两个，因为每个倒斜杠需要由另一个倒斜杠字符来转义）。如果我在OS X或Linux上调用这个函数，该字符串就会是'usr/bin/spam'。

如果需要创建文件名称的字符串，os.path.join()函数就很有用。这些字符串将传递给几个文件相关的函数，本章将进行介绍。例如，下面的例子将一个文件名列表中的名称，添加到文件夹名称的末尾。

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

8.1.2 当前工作目录

每个运行在计算机上的程序，都有一个“当前工作目录”，或cwd。所有没有从根文件夹开始的文件名或路径，都假定在当前工作目录下。利用os.getcwd()函数，可以取得当前工作路径的字符串，并可以利用os.chdir()改变它。在交互式环境中输入以下代码：

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

这里，当前工作目录设置为C:\Python34，所以文件名project.docx指

向C:\Python34\project.docx。如果我们将当前工作目录改为C:\Windows，文件就被解释为C:\Windows\project.docx。

如果要更改的当前工作目录不存在，Python就会显示一个错误。

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "< pyshell#18>", line 1, in < module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

> 注意
>
> 虽然文件夹是目录的更新的名称，但请注意，当前工作目录（或当前目录）是标准术语，没有当前工作文件夹这种说法。

8.1.3 绝对路径与相对路径

有两种方法指定一个文件路径。

- “绝对路径”，总是从根文件夹开始。
- “相对路径”，它相对于程序的当前工作目录。

还有点（.）和点点（..）文件夹。它们不是真正的文件夹，而是可以在路径中使用的特殊名称。单个的句点（“点”）用作文件夹目名称时，是“这个目录”的缩写。两个句点（“点点”）意思是父文件夹。

图8-2是一些文件夹和文件的例子。如果当前工作目录设置为C:\bacon，这些文件夹和文件的相对目录，就设置为图8-2所示的样子。

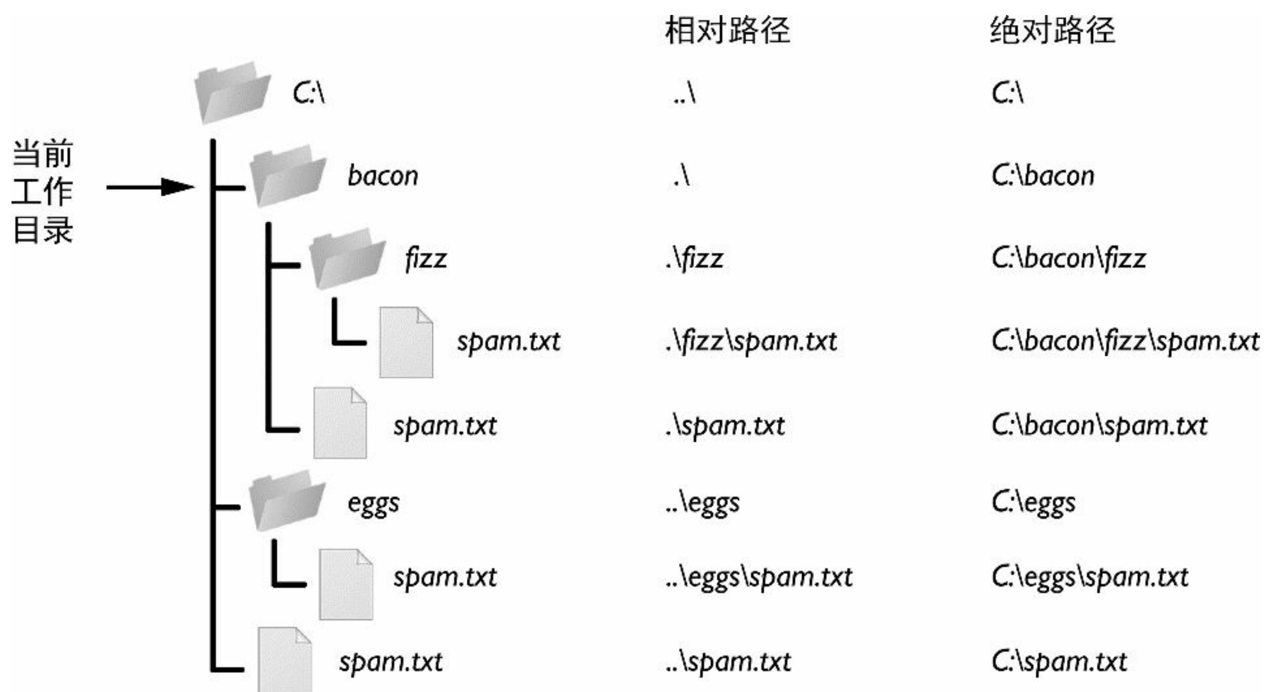


图8-2 在工作目录C:\bacon中的文件夹和文件的相对路径

相对路径开始处的.\是可选的。例如，.\spam.txt和spam.txt指的是同一个文件。

8.1.4 用**os.makedirs()**创建新文件夹

程序可以用os.makedirs()函数创建新文件夹（目录）。在交互式环境中输入以下代码：

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

这不仅将创建C:\delicious文件夹，也会在C:\delicious下创建walnut文件夹，并在C:\delicious\walnut中创建waffles文件夹。也就是说，os.makedirs()将创建所有必要的中间文件夹，目的是确保完整路径名存在。图 8-3 展示了这个文件夹的层次结构。

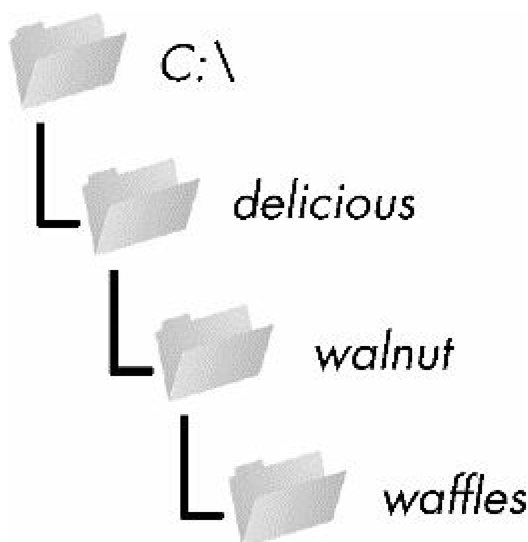


图8-3 os.makedirs('C:\delicious\walnut\waffles')的结果

8.1.5 os.path模块

os.path模块包含了许多与文件名和文件路径相关的有用函数。例如，你已经使用了os.path.join()来构建所有操作系统上都有效的路径。因为os.path是os模块中的模块，所以只要执行import os就可以导入它。如果你的程序需要处理文件、文件夹或文件路径，就可以参考本节中这些简短的例子。os.path模块的完整文档在Python网站上：http://docs.python.org/3/library/os.path.html 。

> 注意
>
> 本章后面的大多数例子都需要os模块，所以要记得在每个脚本开始处导入它，或在重新启动IDLE时导入它。否则，就会遇到错误消息NameError: name 'os' is not defined。

8.1.6 处理绝对路径和相对路径

os.path模块提供了一些函数，返回一个相对路径的绝对路径，以及检查给定的路径是否为绝对路径。

- 调用os.path.abspath(path)将返回参数的绝对路径的字符串。这是将相对路径转换为绝对路径的简便方法。
- 调用os.path.isabs(path)，如果参数是一个绝对路径，就返回True，如果参数是一个相对路径，就返回False。
- 调用os.path.relpath(path, start)将返回从start路径到path的相对路径的字符串。如果没有提供start，就使用当前工作目录作为开始路径。

在交互式环境中尝试以下函数：

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

因为在os.path.abspath()调用时，当前目录是C:\Python34，所以“点”文件夹指的是绝对路径'C:\Python34'。

注意

因为在你的系统上，文件和文件夹可能与我的不同，所以你不能完全遵照本章中的每一个例子。但还是请尝试用你的计算机上存在的文件夹来完成例子。

在交互式环境中，输入以下对os.path.relpath()的调用：

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

调用os.path.dirname(path)将返回一个字符串，它包含path参数中最后一个斜杠之前的所有内容。调用os.path.basename(path)将返回一个字符串，它包含path 参数中最后一个斜杠之后的所有内容。一个路径的目录名称和基本名称如图8-4所示。



图8-4 基本名称跟在路径中最后一个斜杠后，它和文件名一样，
目录名称是最后一个斜杠之前的所有内容

例如，在交互式环境中输入以下代码：

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

如果同时需要一个路径的目录名称和基本名称，就可以调用os.path.split()，获得这两个字符串的元组，像这样：

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
```

```
>>> os.path.split(calcFilePath)

('C:\\Windows\\System32', 'calc.exe')
```

请注意，可以调用os.path.dirname()和os.path.basename()，将它们的返回值放在一个元组中，从而得到同样的元组。

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))

('C:\\Windows\\System32', 'calc.exe')
```

但如果需要两个值，os.path.split()是很好的快捷方式。

同时也请注意，os.path.split()不会接受一个文件路径并返回每个文件夹的字符串的列表。如果需要这样，请使用split()字符串方法，并根据os.path.sep中的字符串进行分割。回忆一下，根据程序运行的计算机，os.path.sep变量设置为正确的文件夹分割斜杠。

例如，在交互式环境中输入以下代码：

```
>>> calcFilePath.split(os.path.sep)
```

```
['C:', 'Windows', 'System32', 'calc.exe']
```

在OS X和Linux系统上，返回的列表头上有一个空字符串：

```
>>> '/usr/bin'.split(os.path.sep)
['', 'usr', 'bin']
```

split()字符串方法将返回一个列表，包含该路径的所有部分。如果向它传递os.path.sep，就能在所有操作系统上工作。

8.1.7 查看文件大小和文件夹内容

一旦有办法处理文件路径，就可以开始搜集特定文件和文件夹的信息。os.path模块提供了一些函数，用于查看文件的字节数以及给定文件夹中的文件和子文件夹。

- 调用os.path.getsize(path)将返回path参数中文件的字节数。
- 调用os.listdir(path)将返回文件名字符串的列表，包含path参数中的每个文件（请注意，这个函数在os模块中，而不是os.path）。

下面是我在交互式环境中尝试这些函数的结果：

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
```

```
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--_snip_--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

可以看到，我的计算机上的calc.exe程序是776192字节。在我的C:\Windows\ system32下有许多文件。如果想知道这个目录下所有文件的总字节数，就可以同时使用os.path.getsize()和os.listdir()。

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
     totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\Sys
```

```
>>> print(totalSize)

1117846456
```

当循环遍历C:\Windows\System32文件夹中的每个文件时，totalSize变量依次增加每个文件的字节数。请注意，我在调用os.path.getsize()时，使用了os.path.join()来连接文件夹名称和当前的文件名。os.path.getsize()返回的整数添加到totalSize中。在循环遍历所有文件后，我打印出totalSize，看看C:\Windows\System32文件夹的总字节数。

8.1.8 检查路径有效性

如果你提供的路径不存在，许多Python函数就会崩溃并报错。os.path模块提供了一些函数，用于检测给定的路径是否存在，以及它是文件还是文件夹。

- 如果path参数所指的文件或文件夹存在，调用os.path.exists(path)将返回True，否则返回False。
- 如果path参数存在，并且是一个文件，调用os.path.isfile(path)将返回True，否则返回False。
- 如果path参数存在，并且是一个文件夹，调用os.path.isdir(path)将返回True，否则返回False。

下面是我在交互式环境中尝试这些函数的结果：

```
>>> os.path.exists('C:\\Windows')
```

```
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

利用os.path.exists()函数，可以确定DVD或闪存盘当前是否连在计算机上。例如，如果在Windows计算机上，我想用卷名D:\检查一个闪存

盘，可以这样做：

```
>>> os.path.exists('D:\\')
False
```

不好！看起来我忘记插入闪存盘了。

8.2 文件读写过程

在熟悉了处理文件夹和相对路径后，你就可以指定文件的位置，进行读写。接下来几节介绍的函数适用于纯文本文件。“纯文本文件”只包含基本文本字符，不包含字体、大小和颜色信息。带有.txt扩展名的文本文件，以及带有.py扩展名的Python脚本文件，都是纯文本文件的例子。它们可以被Windows的Notepad或OS X的TextEdit应用打开。你的程序可以轻易地读取纯文本文件的内容，将它们作为普通的字符串值。

“二进制文件”是所有其他文件类型，诸如字处理文档、PDF、图像、电子表格和可执行程序。如果用Notepad或TextEdit打开一个二进制文件，它看起来就像乱码，如图8-5所示。

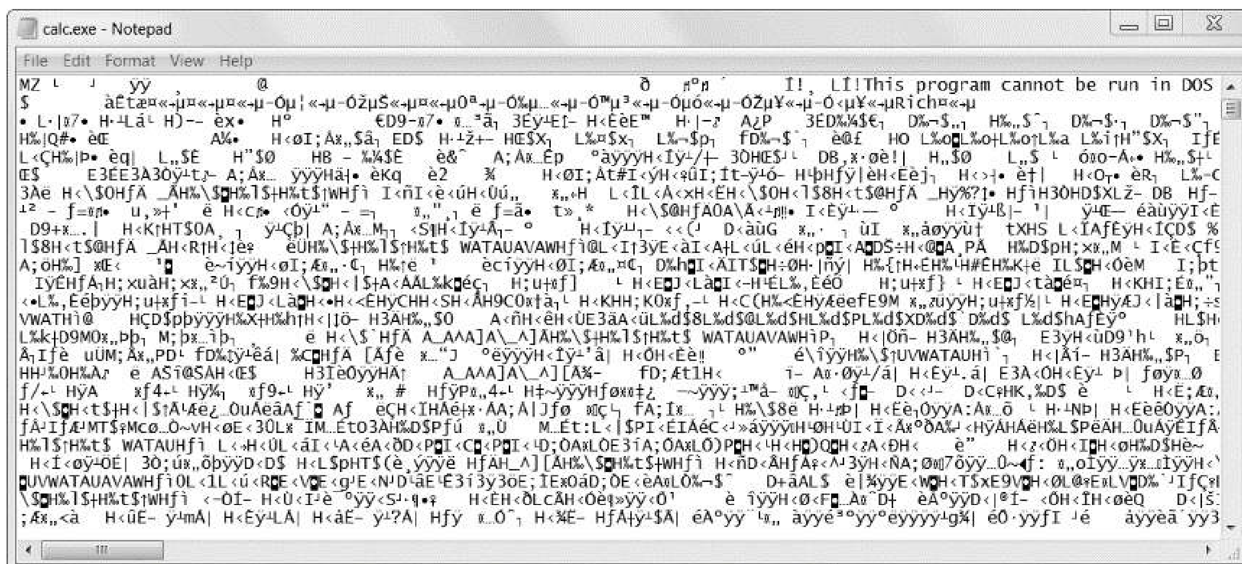


图8-5 在Notepad中打开Windows的calc.exe程序

既然每种不同类型的二进制文件，都必须用它自己的方式来处理，本书就不会探讨直接读写二进制文件。好在，许多模块让二进制文件的处理变得更容易。在本章稍后，你将探索其中一个模块：shelve。

在Python中，读写文件有3个步骤：

1．调用open()函数，返回一个File对象。

2．调用File对象的read()或write()方法。

3．调用File对象的close()方法，关闭该文件。

8.2.1 用**open()**函数打开文件

要用open()函数打开一个文件，就要向它传递一个字符串路径，表明希望打开的文件。这既可以是绝对路径，也可以是相对路径。open()函数返回一个File对象。

尝试一下，先用Notepad或TextEdit创建一个文本文件，名为hello.txt。输入Hello world!作为该文本文件的内容，将它保存在你的用户文件夹中。然后，如果使用Windows，在交互式环境中输入以下代码：

```
>>> helloFile = open('C:\\Users\\_your_home_folder_\\hello.txt')
```

如果使用OS X，在交互式环境中输入以下代码：

```
>>> helloFile = open('/Users/_your_home_folder_/hello.txt')
```

请确保用你自己的计算机用户名取代your_home_folder。例如，我的用户名是asweigart，所以我在windows下输入'C:\Users\asweigart\hello.txt'。

这些命令都将以读取纯文本文件的模式打开文件，或简称为“读模式”。当文件以读模式打开时，Python只让你从文件中读取数据，你不能以任何方式写入或修改它。在Python中打开文件时，读模式是默认的模式。但如果你不希望依赖于Python的默认值，也可以明确指明该模式，向open()传入字符串'r'，作为第二个参数。所以open('/Users/asweigart/hello.txt', 'r')和open('/Users/asweigart/hello.txt')做的事情一样。

调用open()将返回一个File对象。File对象代表计算机中的一个文件，它只是Python中另一种类型的值，就像你已熟悉的列表和字典。在前面的例子中，你将File对象保存在helloFile变量中。现在，当你需要读取或写入该文件，就可以调用helloFile变量中的File对象的方法。

8.2.2 读取文件内容

既然有了一个File对象，就可以开始从它读取内容。如果你希望将整个文件的内容读取为一个字符串值，就使用File对象的read()方法。让我们继续使用保存在helloFile中的hello.txt File对象。在交互式环境中输入以下代码：

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

如果你将文件的内容看成是单个大字符串，read()方法就返回保存在该文件中的这个字符串。

或者，可以使用readlines()方法，从该文件取得一个字符串的列表。列表中的每个字符串就是文本中的每一行。例如，在hello.txt文件相同的目录下，创建一个名为sonnet29.txt的文件，并在其中写入以下文本：

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

确保用换行分开这4行。然后在交互式环境中输入以下代码：

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

请注意，每个字符串值都以一个换行字符\n结束。除了文件的最后一行。与单个大字符串相比，字符串的列表通常更容易处理。

8.2.3 写入文件

Python允许你将内容写入文件，方式与print()函数将字符串"写"到屏幕上类似。但是，如果打开文件时用读模式，就不能写入文件。你需要以"写入纯文本模式"或"添加纯文本模式"打开该文件，或简称为"写模式"和"添加模式"。

写模式将覆写原有的文件，从头开始，就像你用一个新值覆写一个变量的值。将'w'作为第二个参数传递给open()，以写模式打开该文件。不同的是，添加模式将在已有文件的末尾添加文本。你可以认为这类似向一个变量中的列表添加内容，而不是完全覆写该变量。将'a'作为第二个参数传递给open()，以添加模式打开该文件。

如果传递给 open()的文件名不存在，写模式和添加模式都会创建一个新的空文件。在读取或写入文件后，调用close()方法，然后才能再次

打开该文件。

让我们整合这些概念。在交互式环境中输入以下代码：

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
```

```
>>> content = baconFile.read()

>>> baconFile.close()

>>> print(content)

Hello world!
Bacon is not a vegetable.
```

首先，我们以写模式打开bacon.txt。因为还没有bacon.txt，Python就创建了一个。在打开的文件上调用write()，并向write()传入字符串参数'Hello world! \n'，将字符串写入文件，并返回写入的字符个数，包括换行符。然后关闭该文件。

为了将文本添加到文件已有的内容，而不是取代我们刚刚写入的字符串，我们就以添加模式打开该文件。向该文件写入'Bacon is not a vegetable.'，并关闭它。最后，为了将文件的内容打印到屏幕上，我们以默认的读模式打开该文件，调用read()，将得到的内容保存在content中，关闭该文件，并打印content。

请注意，write()方法不会像print()函数那样，在字符串的末尾自动添加换行字符。必须自己添加该字符。

8.3 用shelve模块保存变量

利用shelve模块，你可以将Python程序中的变量保存到二进制的shelf文件中。这样，程序就可以从硬盘中恢复变量的数据。shelve模块让你在程序中添加“保存”和“打开”功能。例如，如果运行一个程序，并输入了一些配置设置，就可以将这些设置保存到一个shelf文件，然后让程序下一次运行时加载它们。

在交互式环境中输入以下代码：

```
>>> import shelve

>>> shelfFile = shelve.open('mydata')

>>> cats = ['Zophie', 'Pooka', 'Simon']

>>> shelfFile['cats'] = cats

>>> shelfFile.close()
```

要利用shelve模块读写数据，首先要导入它。调用函数shelve.open()并传入一个文件名，然后将返回的值保存在一个变量中。可以对这个变量的shelf值进行修改，就像它是一个字典一样。当你完成时，在这个值上调用close()。这里，我们的shelf值保存在shelfFile中。我们创建了一个列表cats，并写下shelfFile['cats'] =cats，将该列表保存在shelfFile中，作为键'cats'关联的值（就像在字典中一样）。然后我们在shelfFile上调用close()。

在Windows上运行前面的代码，你会看到在当前工作目录下有3个新文件：mydata.bak、mydata.dat和mydata.dir。在OS X上，只会创建一个mydata.db文件。

这些二进制文件包含了存储在shelf中的数据。这些二进制文件的格式并不重要，你只需要知道shelve模块做了什么，而不必知道它是怎么做的。该模块让你不用操心如何将程序的数据保存到文件中。

你的程序稍后可以使用shelve模块，重新打开这些文件并取出数据。shelf值不必用读模式或写模式打开，因为它们在打开后，既能读又能写。在交互式环境中输入以下代码：

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
< class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
```

```
>>> shelfFile.close()
```

这里，我们打开了shelf文件，检查我们的数据是否正确存储。输入shelfFile['cats']将返回我们前面保存的同一个列表，所以我们就知道该列表得到了正确存储，然后我们调用close()。

就像字典一样，shelf值有keys()和values()方法，返回shelf中键和值的类似列表的值。因为这些方法返回类似列表的值，而不是真正的列表，所以应该将它们传递给list()函数，取得列表的形式。在交互式环境中输入以下代码：

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

创建文件时，如果你需要在Notepad或TextEdit这样的文本编辑器中读取它们，纯文本就非常有用。但是，如果想要保存Python程序中的数据，那就使用shelve模块。

8.4 用pprint.pformat()函数保存变量

回忆一下5.2节“漂亮打印”中，pprint.pprint()函数将列表或字典中的内容“漂亮打印”到屏幕，而pprint.pformat()函数将返回同样的文本字符串，但不是打印它。这个字符串不仅是易于阅读的格式，同时也是语法上正确的Python代码。假定你有一个字典，保存在一个变量中，你希望保存这个变量和它的内容，以便将来使用。pprint.pformat()函数将提供一个字符串，你可以将它写入.py文件。该文件将成为你自己的模块，如果你需要使用存储在其中的变量，就可以导入它。

例如，在交互式环境中输入以下代码：

```
>>> import pprint

>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc':

>>> pprint.pformat(cats)
```

```
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

这里，我们导入了pprint，以便能使用pprint.pformat()。我们有一个字典的列表，保存在变量cats中。为了让cats中的列表在关闭交互式环境后仍然可用，我们利用pprint.pformat()，将它返回为一个字符串。当我们有了cats中数据的字符串形式，就很容易将该字符串写入一个文件，我们将它命名为myCats.py。

import语句导入的模块本身就是Python脚本。如果来自pprint.pformat()的字符串保存为一个.py文件，该文件就是一个可以导入的模块，像其他模块一样。

由于Python脚本本身也是带有.py文件扩展名的文本文件，所以你的Python程序甚至可以生成其他Python程序。然后可以将这些文件导入到脚本中。

```
>>> import myCats
```

```
>>> myCats.cats
[{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> myCats.cats[0]
{'name': 'Zophie', 'desc': 'chubby'}
>>> myCats.cats[0]['name']
'Zophie'
```

创建一个.py文件（而不是利用shelve模块保存变量）的好处在于，因为它是一个文本文件，所以任何人都可以用一个简单的文本编辑器读取和修改该文件的内容。但是，对于大多数应用，利用shelve模块来保存数据，是将变量保存到文件的最佳方式。只有基本数据类型，诸如整型、浮点型、字符串、列表和字典，可以作为简单文本写入一个文件。例如，File对象就不能够编码为文本。

8.5 项目：生成随机的测验试卷文件

假如你是一位地理老师，班上有35名学生，你希望进行美国各州首府的一个小测验。不妙的是，班里有几个坏蛋，你无法确信学生不会作弊。你希望随机调整问题的次序，这样每份试卷都是独一无二的，这让任何人都不能从其他人那里抄袭答案。当然，手工完成这件事又费时又

无聊。好在，你懂一些Python。

下面是程序所做的事：

- 创建35份不同的测验试卷。
- 为每份试卷创建50个多重选择题，次序随机。
- 为每个问题提供一个正确答案和3个随机的错误答案，次序随机。
- 将测验试卷写到35个文本文件中。
- 将答案写到35个文本文件中。

这意味着代码需要做下面的事：

- 将州和它们的首府保存在一个字典中。
- 针对测验文本文件和答案文本文件，调用open()、write()和close()。
- 利用random.shuffle()随机调整问题和多重选项的次序。

第1步：将测验数据保存在一个字典中

第一步是创建一个脚本框架，并填入测验数据。创建一个名为randomQuiz Generator.py的文件，让它看起来像这样：

```
  #! python3
  # randomQuizGenerator.py - Creates quizzes with questions and answers in
  # random order, along with the answer key.

❶ import random

  # The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoe
  'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denve
  'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
  'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
  'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas':
  'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine':
  'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan'
  'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri
  'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada':
  'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New
  Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
  'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma Cit
  'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Provide
  'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
```

```
'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

# Generate 35 quiz files.
❸ for quizNum in range(35):
    # TODO: Create the quiz and answer key files.

    # TODO: Write out the header for the quiz.

    # TODO: Shuffle the order of the states.

    # TODO: Loop through all 50 states, making a question for each.
```

因为这个程序将随机安排问题和答案的次序，所以需要导入random模块❶，以便利用其中的函数。capitals变量❷含一个字典，以美国州名作为键，以州首府作为值。因为你希望创建35份测验试卷，所以实际生成测验试卷和答案文件的代码（暂时用TODO注释标注）会放在一个for循环中，循环35次❸（这个数字可以改变，生成任何数目的测验试卷文件）。

第**2**步：创建测验文件，并打乱问题的次序

现在是时候填入那些TODO了。

循环中的代码将重复执行35次（每次生成一份测验试卷），所以在循环中，你只需要考虑一份测验试卷。首先你要创建一个实际的测验试卷文件，它需要有唯一的文件名，并且有某种标准的标题部分，留出位置，让学生填写姓名、日期和班级。然后需要得到随机排列的州的列表，稍后将用它来创建测验试卷的问题和答案。

在randomQuizGenerator.py中添加以下代码行：

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--_snip_--
```

```
# Generate 35 quiz files.
for quizNum in range(35):
    # Create the quiz and answer key files.
❶     quizFile = open('capitalsquiz%s.txt' % (quizNum + 1), 'w')
❷     answerKeyFile = open('capitalsquiz_answers%s.txt' % (quizNum + 1), 'w
    # Write out the header for the quiz.
❸     quizFile.write('Name:\n\nDate:\n\nPeriod:\n\n')
    quizFile.write((' ' * 20) + 'State Capitals Quiz (Form %s)' % (quizNu
    quizFile.write('\n\n')
```

```
    # Shuffle the order of the states.
    states = list(capitals.keys())
❹    random.shuffle(states)

# TODO: Loop through all 50 states, making a question for each.
```

测验试卷的文件名将是capitalsquiz<N>.txt，其中<N>是该测验试卷的唯一编号，来自于quizNum，即for循环的计数器。针对

capitalsquiz<N>.txt的答案将保存在一个文本文件中，名为capitalsquiz_answers<N>.txt。每次执行循环，'capitalsquiz%s.txt'和'capitalsquiz_answers%s.txt'中的占位符%s都将被(quizNum + 1)取代，所以第一个测验试卷和答案将是capitalsquiz1.txt和capitalsquiz_answers1.txt。在❶和❷的open()函数调用将创建这些文件，以'w'作为第二个参数，以写模式打开它们。

❸处write()语句创建了测验标题，让学生填写。最后，利用random.shuffle()函数❹，创建了美国州名的随机列表。该函数重新随机排列传递给它的列表中的值。

第3步：创建答案选项

现在需要为每个问题生成答案选项，这将是A到D的多重选择。你需要创建另一个for循环，该循环生成测验试卷的50个问题的内容。然后里面会嵌套第三个for循环，为每个问题生成多重选项。让你的代码看起来像这样：

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.

--_snip_--

    # Loop through all 50 states, making a question for each.




    for questionNum in range(50):










        # Get right and wrong answers.
```

```
❶         correctAnswer = capitals[states[questionNum]]
❷         wrongAnswers = list(capitals.values())
❸         del wrongAnswers[wrongAnswers.index(correctAnswer)]
❹         wrongAnswers = random.sample(wrongAnswers, 3)
❺         answerOptions = wrongAnswers + [correctAnswer]
❻         random.shuffle(answerOptions)

          # TODO: Write the question and answer options to the quiz file.
```

```
    # TODO: Write the answer key to a file.
```

正确的答案很容易得到，它作为一个值保存在capitals字典中❶。这个循环将遍历打乱过的states列表中的州，从states[0]到states[49]，在capitals中找到每个州，将该州对应的首府保存在correctAnswer中。

可能的错误答案列表需要一点技巧。你可以从capitals字典中复制所有的值❷，删除正确的答案❸，然后从该列表中选择3个随机的值❹。random.sample()函数使得这种选择很容易，它的第一个参数是你希望选择的列表，第二个参数是你希望选择的值的个数。完整的答案选项列表是这3个错误答案与正确答案的组合❺。最后，答案需要随机排列❻，这样正确的答案就不会总是选项D。

第4步：将内容写入测验试卷和答案文件

剩下来就是将问题写入测验试卷文件，将答案写入答案文件。让你的代码看起来像这样：

```
#! python3
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key.
```

```
  --_snip_--

    # Loop through all 50 states, making a question for each.
    for questionNum in range(50):
      --_snip_--

      # Write the question and the answer options to the quiz file.
      quizFile.write('%s. What is the capital of %s?\n' % (questionNum + 1,
          states[questionNum]))
❶     for i in range(4):
❷         quizFile.write(' %s. %s\n' % ('ABCD'[i], answerOptions[i]))
      quizFile.write('\n')
```

```
    # Write the answer key to a file.

❸    answerKeyFile.write('%s. %s\n' % (questionNum + 1, 'ABCD'[

        answerOptions.index(correctAnswer)]))

    quizFile.close()

    answerKeyFile.close()
```

一个遍历整数0到3的for循环，将答案选项写入answerOptions列表❶。❷处的表达式'ABCD'[i]将字符串'ABCD'看成是一个数组，它在循环的每次迭代中，将分别求值为'A'、'B'、'C'和'D'。

在最后一行❸，表达式answerOptions.index(correctAnswer)将在随机排序的答案选项中，找到正确答案的整数下标，并且'ABCD'[answerOptions.index(correctAnswer)]将求值为正确答案的字母，写入到答案文件中。

在运行该程序后，下面就是capitalsquiz1.txt文件看起来的样子。但是，你的问题和答案选项当然与这里显示的可能会不同。这取决于random.shuffle()调用的结果：

```
Name:

Date:

Period:
                    State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?

    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln

--snip




--
```

对应的capitalsquiz_answers1.txt文本文件看起来像这样：

```
1. D

2. C
3. A
4. C
--snip
```

```
--
```

8.6 项目：多重剪贴板

假定你有一个无聊的任务，要填充一个网页或软件中的许多表格，其中包含一些文本字段。剪贴板让你不必一次又一次输入同样的文本，但剪贴板上一次只有一个内容。如果你有几段不同的文本需要拷贝粘贴，就不得不一次又一次的标记和拷贝几个同样的内容。

可以编写一个Python程序，追踪几段文本。这个“多重剪贴板”将被命名为mcb.pyw（因为“mcb”比输入“multiclipboard”更简单）。.pyw扩展名意味着Python运行该程序时，不会显示终端窗口（详细内容请参考附录B）。

该程序将利用一个关键字保存每段剪贴板文本。例如，当运行py mcb.pyw save spam，剪贴板中当前的内容就用关键字spam保存。通过运行py mcb.pyw spam，这段文本稍后将重新加载到剪贴板中。如果用户忘记了都有哪些关键字，他们可以运行py mcb.pyw list，将所有关键字的列表复制到剪贴板中。

下面是程序要做的事：

- 针对要检查的关键字，提供命令行参数。
- 如果参数是save，那么将剪贴板的内容保存到关键字。
- 如果参数是list，就将所有的关键字拷贝到剪贴板。
- 否则，就将关键词对应的文本拷贝到剪贴板。

这意味着代码需要做下列事情：

- 从sys.argv读取命令行参数。
- 读写剪贴板。

- 保存并加载shelf文件。

如果你使用Windows，可以创建一个名为mcb.bat的批处理文件，很容易地通过“Run…”窗口运行这个脚本。该批处理文件包含如下内容：

```
@pyw.exe C:\Python34\mcb.pyw %*
```

第1步：注释和shelf设置

我们从一个脚本框架开始，其中包含一些注释和基本设置。让你的代码看起来像这样：

```
  #! python3
  # mcb.pyw - Saves and loads pieces of text to the clipboard.
❶ # Usage: py.exe mcb.pyw save <keyword> - Saves clipboard to keyword.
  #        py.exe mcb.pyw <keyword> - Loads keyword to clipboard.
  #        py.exe mcb.pyw list - Loads all keywords to clipboard.

❷ import shelve, pyperclip, sys

❸ mcbShelf = shelve.open('mcb')

  # TODO: Save clipboard content.

  # TODO: List keywords and load content.

  mcbShelf.close()
```

将一般用法信息放在文件顶部的注释中，这是常见的做法❶。如果忘了如何运行这个脚本，就可以看看这些注释，帮助回忆起来。然后导入模块❷。拷贝和粘贴需要pyperclip模块，读取命令行参数需要sys模块。shelve模块也需要准备好。当用户希望保存一段剪贴板文本时，你需要将它保存到一个shelf文件中。然后，当用户希望将文本拷贝回剪贴板时，你需要打开shelf文件，将它重新加载到程序中。这个shlef文件命

名时带有前缀mcb❸。

第2步：用一个关键字保存剪贴板内容

根据用户希望保存文本到一个关键字，或加载文本到剪贴板，或列出已有的关键字，该程序做的事情不一样。让我们来处理第一种情况。让你的代码看起来像这样：

```
  #! python3
  # mcb.pyw - Saves and loads pieces of text to the clipboard.
  --snip

--

  # Save clipboard content.

❶ if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':

❷         mcbShelf[sys.argv[2]] = pyperclip.paste()

elif len(sys.argv) == 2:

❸     # TODO: List keywords and load content.

  mcbShelf.close()
```

如果第一个命令行参数（它总是在sys.argv列表的下标1处）是字符串'save' ❶，第二个命令行参数就是保存剪贴板当前内容的关键字。关键字将用做 mcbShelf中的键，值就是当前剪贴板上的文本❷。

如果只有一个命令行参数，就假定它要么是'list'，要么是需要加载到剪贴板的关键字。稍后你将实现这些代码。现在只是放上一条TODO注释❸。

第3步：列出关键字和加载关键字的内容

最后，让我们实现剩下的两种情况。用户希望从关键字加载剪贴板文本，或希望列出所有可用的关键字。让你的代码看起来像这样：

```
  #! python3
  # mcb.pyw - Saves and loads pieces of text to the clipboard.
  --snip
--
  # Save clipboard content.
  if len(sys.argv) == 3 and sys.argv[1].lower() == 'save':
          mcbShelf[sys.argv[2]] = pyperclip.paste()
  elif len(sys.argv) == 2:
      # List keywords and load content.
❶     if sys.argv[1].lower() == 'list':
```

```
❷         pyperclip.copy(str(list(mcbShelf.keys())))

    elif sys.argv[1] in mcbShelf:

❸         pyperclip.copy(mcbShelf[sys.argv[1]])

  mcbShelf.close()
```

如果只有一个命令行参数，首先检查它是不是'list' ❶。如果是，表示shelf键的列表的字符串将被拷贝到剪贴板❷。用户可以将这个列表拷贝到一个打开的文本编辑器，进行查看。

否则，你可以假定该命令行参数是一个关键字。如果这个关键字是shelf中的一个键，就可以将对应的值加载到剪贴板❸。

齐活了！加载这个程序有几个不同步骤，这取决于你的计算机使用哪种操作系统。请查看附录B，了解操作系统的详情。

回忆一下第6章中创建的口令保管箱程序，它将口令保存在一个字典中。更新口令需要更改该程序的源代码。这不太理想，因为普通用户不太适应通过更改源代码来更新他们的软件。而且，每次修改程序的源

代码时，就有可能不小心引入新的缺陷。将程序的数据保存在不同的地方，而不是在代码中，就可以让别人更容易使用你的程序，并且更不容易出错。

8.7 小结

文件被组织在文件夹中（也称为目录），路径描述了一个文件的位置。运行在计算机上的每个程序都有一个当前工作目录，它让你相对于当前的位置指定文件路径，而非总是需要完整路径（绝对路径）。os.path模块包含许多函数，用于操作文件路径。

你的程序也可以直接操作文本文件的内容。open()函数将打开这些文件，将它们的内容读取为一个大字符串（利用reae()方法），或读取为字符串的列表（利用方法readlines()）。Open()函数可以将文件以写模式或添加模式打开，分别创建新的文本文件或在原有的文本文件中添加内容。

在前面几章中，你利用剪贴板在程序中获得大量文本，而不是通过手工输入。现在你可以用程序直接读取硬盘上的文件，这是一大进步。因为文件比剪贴板更不易变化。在下一章中，你将学习如何处理文件本身，包括复制、删除、重命名、移动等。

8.8 习题

1．相对路径是相对于什么？

2．绝对路径从什么开始？

3．os.getcwd()和os.chdir()函数做什么事？

4．.和..文件夹是什么？

5．在C:\bacon\eggs\spam.txt中，哪一部分是目录名称，哪一部分是基本名称？

6．可以传递给open()函数的3种“模式”参数是什么？

7．如果已有的文件以写模式打开，会发生什么？

8．read()和readlines()方法之间的区别是什么？

9．shelf值与什么数据结构相似？

8.9 实践项目

作为实践，设计并编写下列程序。

8.9.1 扩展多重剪贴板

扩展本章中的多重剪贴板程序，增加一个delete <keyword>命令行参数，它将从shelf中删除一个关键字。然后添加一个delete命令行参数，它将删除所有关键字。

8.9.2 疯狂填词

创建一个疯狂填词（Mad Libs）程序，它将读入文本文件，并让用户在该文本文件中出现ADJECTIVE、NOUN、ADVERB或VERB等单词的地方，加上他们自己的文本。例如，一个文本文件可能看起来像这样：

```
The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was
unaffected by these events.
```

程序将找到这些出现的单词，并提示用户取代它们。

```
Enter an adjective:
silly
```

```
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck
```

以下的文本文件将被创建：

```
The silly panda walked to the chandelier and then screamed. A nearby pickup
truck was unaffected by these events.
```

结果应该打印到屏幕上，并保存为一个新的文本文件。

8.9.3 正则表达式查找

编写一个程序，打开文件夹中所有的.txt文件，查找匹配用户提供的正则表达式的所有行。结果应该打印到屏幕上。

第9章　组织文件

在上一章中，你学习了如何用Python创建并写入新文件。你的程序也可以组织硬盘上已经存在的文件。也许你曾经经历过查找一个文件夹，里面有几十个、几百个，甚至上千个文件，需要手工进行复制、改名、移动或压缩。或者考虑下面这样的任务：

- 在一个文件夹及其所有子文件夹中，复制所有的pdf文件（且只复制pdf文件）
- 针对一个文件夹中的所有文件，删除文件名中前导的零，该文件夹中有数百个文件，名为spam001.txt、spam002.txt、spam003.txt等。
- 将几个文件夹的内容压缩到一个ZIP文件中（这可能是一个简单的备份系统）

所有这种无聊的任务，正是在请求用Python实现自动化。通过对电脑编程来完成这些任务，你就把它变成了一个快速工作的文件职员，而且从不犯错。

在开始处理文件时你会发现，如果能够很快查看文件的扩展名（.txt、.pdf、.jpg等），是很有帮助的。在OS X和Linux上，文件浏览器很有可能自动显示扩展名。在Windows上，文件扩展名可能默认是隐藏的。要显示扩展名，请点开Start►Control Panel►Appearance和Personalization►Folder选项。在View选项卡中，Advanced Settings之下，取消Hide extensions for known file types复选框。

9.1 shutil模块

shutil（或称为shell工具）模块中包含一些函数，让你在Python程序中复制、移动、改名和删除文件。要使用shutil的函数，首先需要import shutil。

9.1.1 复制文件和文件夹

shutil模块提供了一些函数，用于复制文件和整个文件夹。

调用shutil.copy(source, destination)，将路径source处的文件复制到路径destination处的文件夹（source和destination都是字符串）。如果destination是一个文件名，它将作为被复制文件的新名字。该函数返回一个字符串，表示被复制文件的路径。

在交互式环境中输入以下代码，看看shutil.copy()的效果：

```
  >>> import shutil, os
  >>> os.chdir('C:\\')
❶ >>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
  'C:\\delicious\\spam.txt'
❷ >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
  'C:\\delicious\\eggs2.txt'
```

第一个shutil.copy()调用将文件C:\spam.txt复制到文件夹C:\delicious。返回值是刚刚被复制的文件的路径。请注意，因为指定了一个文件夹作为目的地❶，原来的文件名spam.txt就被用作新复制的文件名。第二个shutil.copy()调用❷也将文件C:\eggs.txt复制到文件夹C:\delicious，但为新文件提供了一个名字eggs2.txt。

shutil.copy()将复制一个文件，shutil.copytree()将复制整个文件夹，以及它包含的文件夹和文件。调用shutil.copytree(source, destination)，将路径source处的文件夹，包括它的所有文件和子文件夹，复制到路径destination处的文件夹。source和destination参数都是字符串。该函数返回一个字符串，是新复制的文件夹的路径。

在交互式环境中输入以下代码：

```
>>> import shutil, os

>>> os.chdir('C:\\')

>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')

'C:\\bacon_backup'
```

shutil.copytree()调用创建了一个新文件夹，名为bacon_backup，其中的内容与原来的bacon文件夹一样。现在你已经备份了非常非常宝贵的“bacon”。

9.1.2 文件和文件夹的移动与改名

调用shutil.move(source, destination)，将路径source处的文件夹移动到路径destination，并返回新位置的绝对路径的字符串。

如果destination指向一个文件夹，source文件将移动到destination中，并保持原来的文件名。例如，在交互式环境中输入以下代码：

```
>>> import shutil

>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')

'C:\\eggs\\bacon.txt'
```

假定在C:\目录中已存在一个名为eggs的文件夹，这个shutil.move()调用就是说，“将C:\bacon.txt移动到文件夹C:\eggs中。

如果在C:\eggs中原来已经存在一个文件bacon.txt，它就会被覆写。因为用这种方式很容易不小心覆写文件，所以在使用move()时应该注意。

destination路径也可以指定一个文件名。在下面的例子中，source文件被移动并改名。

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')

'C:\\eggs\\new_bacon.txt'
```

这一行是说，“将C:\bacon.txt移动到文件夹C:\eggs，完成之后，将bacon.txt文件改名为new_bacon.txt。”

前面两个例子都假设在C:\目录下有一个文件夹eggs。但是如果没有eggs文件夹，move()就会将bacon.txt改名，变成名为eggs的文件。

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

这里，move()在C:\目录下找不到名为eggs的文件夹，所以假定destination指的是一个文件，而非文件夹。所以bacon.txt文本文件被改名为eggs（没有.txt文件扩展名的文本文件），但这可能不是你所希望的！这可能是程序中很难发现的缺陷，因为move()调用会很开心地做一些事情，但和你所期望的完全不同。这也是在使用move()时要小心的另一个理由。

最后，构成目的地的文件夹必须已经存在，否则Python会抛出异常。在交互式环境中输入以下代码：

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Python34\lib\shutil.py", line 521, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'
```

```
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "< pyshell#29>", line 1, in < module>
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
  File "C:\Python34\lib\shutil.py", line 533, in move
    copy2(src, real_dst)
  File "C:\Python34\lib\shutil.py", line 244, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
  File "C:\Python34\lib\shutil.py", line 108, in copyfile
    with open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist
eggs\\ham'
```

Python在does_not_exist目录中寻找eggs和ham。它没有找到不存在的目录，所以不能将spam.txt移动到指定的路径。

9.1.3 永久删除文件和文件夹

利用os模块中的函数，可以删除一个文件或一个空文件夹。但利用shutil模块，可以删除一个文件夹及其所有的内容。

- 用os.unlink(path)将删除path处的文件。
- 调用os.rmdir(path)将删除path处的文件夹。该文件夹必须为空，其中没有任何文件和文件夹。
- 调用shutil.rmtree(path)将删除path处的文件夹，它包含的所有文件和文件夹都会被删除。

在程序中使用这些函数时要小心！可以第一次运行程序时，注释掉这些调用，并且加上print()调用，显示会被删除的文件。这样做是一个好主意。下面有一个Python程序，本来打算删除具有.txt扩展名的文件，但有一处录入错误（用粗体突出显示），结果导致它删除了.rxt文件。

```
import os
for filename in os.listdir():
    if filename.endswith('.r
```

```
xt'):
        os.unlink(filename)
```

如果你有某些重要的文件以.rxt结尾，它们就会被不小心永久地删除。作为替代，你应该先运行像这样的程序：

```
import os
for filename in os.listdir():
    if filename.endswith('.rxt'):
        #os.unlink(filename)
        print(filename)
```

现在os.unlink()调用被注释掉，所以Python会忽略它。作为替代，你会打印出将被删除的文件名。先运行这个版本的程序，你就会知道，你不小心告诉程序要删除.rxt文件，而不是.txt文件。

在确定程序按照你的意图工作后，删除print(filename)代码行，取消os.unlink(filename)代码行的注释。然后再次运行该程序，实际删除这些文件。

9.1.4 用**send2trash**模块安全地删除

因为Python内建的shutil.rmtree()函数不可恢复地删除文件和文件夹，所以用起来可能有危险。删除文件和文件夹的更好方法，是使用第三方的send2trash模块。你可以在终端窗口中运行pip install send2trash，安装该模块（参见附录A，其中更详细地解释了如何安装第三方模块）。

利用send2trash，比Python常规的删除函数要安全得多，因为它会将

文件夹和文件发送到计算机的垃圾箱或回收站，而不是永久删除它们。如果因程序缺陷而用send2trash删除了某些你不想删除的东西，稍后可以从垃圾箱恢复。

安装send2trash后，在交互式环境中输入以下代码：

```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

一般来说，总是应该使用send2trash.send2trash()函数来删除文件和文件夹。虽然它将文件发送到垃圾箱，让你稍后能够恢复它们，但是这

不像永久删除文件，不会释放磁盘空间。如果你希望程序释放磁盘空间，就要用os和shutil来删除文件和文件夹。请注意，send2trash()函数只能将文件送到垃圾箱，不能从中恢复文件。

9.2 遍历目录树

假定你希望对某个文件夹中的所有文件改名，包括该文件夹中所有子文件夹中的所有文件。也就是说，你希望遍历目录树，处理遇到的每个文件。写程序完成这件事，可能需要一些技巧。好在，Python提供了一个函数，替你处理这个过程。

请看C:\delicious文件夹及其内容，如图9-1所示。

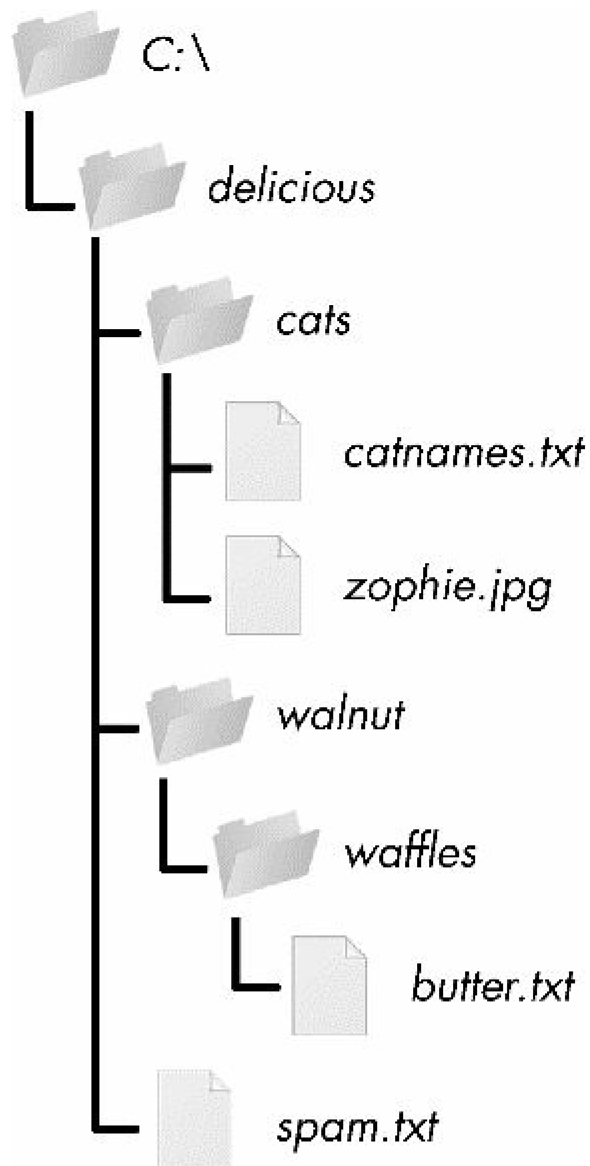


图9-1 一个示例文件夹，包含3个文件夹和4个文件

这里有一个例子程序，针对图9-1的目录树，使用了os.walk()函数：

```
import os

for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)
```

```
    print('')
```

os.walk()函数被传入一个字符串值，即一个文件夹的路径。你可以在一个for循环语句中使用os.walk()函数，遍历目录树，就像使用range()函数遍历一个范围的数字一样。不像range()，os.walk()在循环的每次迭代中，返回3个值：

1. 当前文件夹名称的字符串。

2. 当前文件夹中子文件夹的字符串的列表。

3. 当前文件夹中文件的字符串的列表。

所谓当前文件夹，是指for循环当前迭代的文件夹。程序的当前工作目录，不会因为os.walk()而改变。

就像你可以在代码for i in range(10):中选择变量名称i一样，你也可以选择前面列出来的3个字的变量名称。我通常使用foldername、subfolders和filenames。

运行该程序，它的输出如下：

```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

因为os.walk()返回字符串的列表，保存在subfolder和filename变量中，所以你可以在它们自己的for循环中使用这些列表。用你自己定制的代码，取代print()函数调用（或者如果不需要，就删除for循环）。

9.3 用zipfile模块压缩文件

你可能熟悉ZIP文件（带有.zip文件扩展名），它可以包含许多其他文件的压缩内容。压缩一个文件会减少它的大小，这在因特网上传输时很有用。因为一个ZIP文件可以包含多个文件和子文件夹，所以它是一种很方便的方式，将多个文件打包成一个文件。这个文件叫做“归档文件”，然后可以用作电子邮件的附件，或其他用途。

利用zipfile模块中的函数，Python程序可以创建和打开（或解压）ZIP文件。假定你有一个名为example.zip的zip文件，它的内容如图9-2所示。

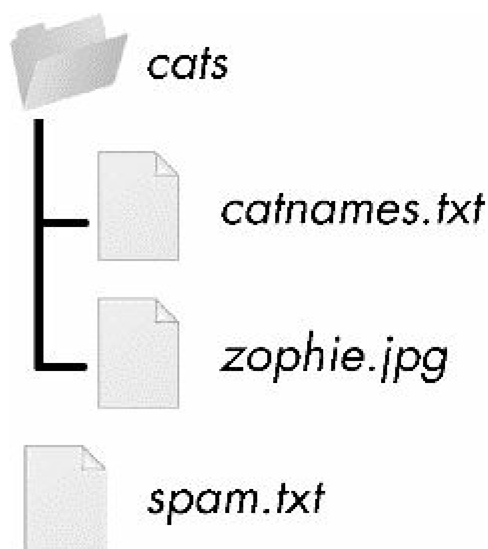


图9-2 example.zip的内容

可以从http://nostarch.com/automatestuff/ 下载这个ZIP文件，或者利用计算机上已有的一个ZIP文件，接着完成下面的操作。

9.3.1 读取ZIP文件

要读取ZIP文件的内容，首先必须创建一个ZipFile对象（请注意大写首字母Z和F）。ZipFile对象在概念上与File对象相似，你在第8章中曾经看到open()函数返回File对象：它们是一些值，程序通过它们与文件打交道。要创建一个ZipFile对象，就调用zipfile.ZipFile()函数，向它传入一个字符串，表示.zip文件的文件名。请注意，zipfile是Python模块的名称，ZipFile()是函数的名称。

例如，在交互式环境中输入以下代码：

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
```

```
13908
>>> spamInfo.compress_size
3828
❶ >>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamI
.compress_size, 2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

ZipFile对象有一个namelist()方法，返回ZIP文件中包含的所有文件和文件夹的字符串的列表。这些字符串可以传递给ZipFile对象的getinfo()方法，返回一个关于特定文件的ZipInfo对象。ZipInfo对象有自己的属性，诸如表示字节数的file_size和compress_size，它们分别表示原来文件大小和压缩后文件大小。ZipFile对象表示整个归档文件，而ZipInfo对象则保存该归档文件中每个文件的有用信息。

❶处的命令计算出example.zip压缩的效率，用压缩后文件的大小除

以原来文件的大小，并以%s字符串格式打印出这一信息。

9.3.2 从ZIP文件中解压缩

ZipFile对象的extractall()方法从ZIP文件中解压缩所有文件和文件夹，放到当前工作目录中。

```
  >>> import zipfile, os
  >>> os.chdir('C:\\') # move to the folder with example.zip
  >>> exampleZip = zipfile.ZipFile('example.zip')
❶ >>> exampleZip.extractall()
  >>> exampleZip.close()
```

运行这段代码后，example.zip的内容将被解压缩到C:\。或者，你可

以向extractall()传递的一个文件夹名称，它将文件解压缩到那个文件夹，而不是当前工作目录。如果传递给extractall()方法的文件夹不存在，它会被创建。例如，如果你用exampleZip.extractall('C:\ delicious')取代❶处的调用，代码就会从example.zip中解压缩文件，放到新创建的C:\delicious文件夹中。

ZipFile对象的extract()方法从ZIP文件中解压缩单个文件。继续交互式环境中的例子：

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

传递给extract()的字符串，必须匹配namelist()返回的字符串列表中的一个。或者，你可以向extract()传递第二个参数，将文件解压缩到指定的文件夹，而不是当前工作目录。如果第二个参数指定的文件夹不存在，Python就会创建它。extract()的返回值是被压缩后文件的绝对路径。

9.3.3 创建和添加到ZIP文件

要创建你自己的压缩ZIP文件，必须以“写模式”打开ZipFile对象，即传入'w'作为第二个参数（这类似于向open()函数传入'w'，以写模式打开一个文本文件）。

如果向ZipFile对象的write()方法传入一个路径，Python就会压缩该路径所指的文件，将它加到ZIP文件中。write()方法的第一个参数是一个字符串，代表要添加的文件名。第二个参数是“压缩类型”参数，它告诉计算机使用怎样的算法来压缩文件。可以总是将这个值设置为zipfile.ZIP_DEFLATED（这指定了deflate压缩算法，它对各种类型的数据都很有效）。在交互式环境中输入以下代码：

```
>>> import zipfile

>>> newZip = zipfile.ZipFile('new.zip', 'w')

>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)

>>> newZip.close()
```

这段代码将创建一个新的ZIP文件，名为new.zip，它包含spam.txt压缩后的内容。

要记住，就像写入文件一样，写模式将擦除ZIP文件中所有原有的内容。如果只是希望将文件添加到原有的ZIP文件中，就要向zipfile.ZipFile()传入'a'作为第二个参数，以添加模式打开ZIP文件。

9.4 项目：将带有美国风格日期的文件改名为欧洲风格日期

假定你的老板用电子邮件发给你上千个文件，文件名包含美国风格的日期（MM-DD-YYYY），需要将它们改名为欧洲风格的日期（DD-MM-YYYY）。手工完成这个无聊的任务可能需要几天时间！让我们写一个程序来完成它。

下面是程序要做的事：

- 检查当前工作目录的所有文件名，寻找美国风格的日期。
- 如果找到，将该文件改名，交换月份和日期的位置，使之成为欧洲风格。

这意味着代码需要做下面的事情：

- 创建一个正则表达式，可以识别美国风格日期的文本模式。
- 调用os.listdir()，找出工作目录中的所有文件。
- 循环遍历每个文件名，利用该正则表达式检查它是否包含日期。
- 如果它包含日期，用shutil.move()对该文件改名。

对于这个项目，打开一个新的文件编辑器窗口，将代码保存为renameDates.py。

第1步：为美国风格的日期创建一个正则表达式

程序的第一部分需要导入必要的模块，并创建一个正则表达式，它能识别MM-DD-YYYY格式的日期。TODO注释将提醒你，这个程序还要写什么。将它们作为TODO，就很容易利用IDLE的Ctrl-F查找功能找到它们。让你的代码看起来像这样：

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
```

```
  # to European DD-MM-YYYY.

❶ import shutil, os, re

  # Create a regex that matches files with the American date format.
❷ datePattern = re.compile(r"""^(.*?)     # all text before the date
      ((0|1)?\d)-                   # one or two digits for the month
      ((0|1|2|3)?\d)-              # one or two digits for the day
      ((19|20)\d\d)               # four digits for the year
      (.*?)$                       # all text after the date
❸     """, re.VERBOSE)

  # TODO: Loop over the files in the working directory.

  # TODO: Skip files without a date.

  # TODO: Get the different parts of the filename.

  # TODO: Form the European-style filename.

  # TODO: Get the full, absolute file paths.

  # TODO: Rename the files.
```

通过本章，你知道shutil.move()函数可以用于文件改名：它的参数是要改名的文件名，以及新的文件名。因为这个函数存在于shutil模块中，所以你必须导入该模块❶。

在为这些文件改名之前，需要确定哪些文件要改名。文件名如果包含spam4-4-1984.txt和01-03-2014eggs.zip这样的日期，就应该改名，而文件名不包含日期的应该忽略，诸如littlebrother.epub。

可以用正则表达式来识别该模式。在开始导入re模块后，调用re.compile()创建一个Regex对象❷。传入re.VERBOSE作为第二参数❸，这将在正则表达式字符串中允许空白字符和注释，让它更可读。

正则表达式字符串以^(.*?)开始，匹配文件名开始处、日期出现之前的任何文本。((0|1)?\d)分组匹配月份。第一个数字可以是0或1，所以正则表达式匹配12，作为十二月份，也会匹配02，作为二月份。这个数

字也是可选的，所以四月份可以是04或4。日期的分组是((0|1|2|3)?\d)，它遵循类似的逻辑。3、03和31是有效的日期数字（是的，这个正则表达式会接受一些无效的日期，诸如4-31-2014、2-29-2013和0-15-2014。日期有许多特例，很容易被遗漏。为了简单，这个程序中的正则表达式已经足够好了）。

虽然1885是一个有效的年份，但你可能只在寻找20世纪和21世纪的年份。这防止了程序不小心匹配非日期的文件名，它们和日期格式类似，诸如10-10-1000.txt。

正则表达式的(.*?)$部分，将匹配日期之后的任何文本。

第**2**步：识别文件名中的日期部分

接下来，程序将循环遍历os.listdir()返回的文件名字符串列表，用这个正则表达式匹配它们。文件名不包含日期的文件将被忽略。如果文件名包含日期，匹配的文本将保存在几个变量中。用下面的代码代替程序中前3个TODO：

```
  #! python3
  # renameDates.py - Renames filenames with American MM-DD-YYYY date format
  # to European DD-MM-YYYY.

  --snip

--

  # Loop over the files in the working directory.

  for amerFilename in os.listdir('.'):
```

```
    mo = datePattern.search(amerFilename)

    # Skip files without a date.
❶   if mo == None:
❷       continue

❸   # Get the different parts of the filename.
    beforePart  = mo.group(1)
```

```
        monthPart   = mo.group(2)

        dayPart     = mo.group(4)

        yearPart     = mo.group(6)

        afterPart   = mo.group(8)

  --snip

--
```

如果search()方法返回的Match对象是None❶，那么amerFilename中的文件名不匹配该正则表达式。continue语句❷将跳过循环剩下的部分，转向下一个文件名。

否则，该正则表达式分组匹配的不同字符串，将保存在名为beforePart、monthPart、dayPart、yearPart和afterPar的变量中❸。这些变量中的字符串将在下一步中使用，用于构成欧洲风格的文件名。

为了让分组编号直观，请尝试从头阅读该正则表达式，每遇到一个左括号就计数加一。不要考虑代码，只是写下该正则表达式的框架。这有助于使分组变得直观，例如：

```
datePattern = re.compile(r"""^(1




)     # all text before the date
    (2




(3




) )-                         # one or two digits for the month
    (4




(5




) )-                         # one or two digits for the day
    (6




(7
```

```
) )                          # four digits for the year
    (8

)$                              # all text after the date
    """, re.VERBOSE)
```

这里，编号1至8代表了该正则表达式中的分组。写出该正则表达式的框架，其中只包含括号和分组编号。这让你更清楚地理解所写的正则表达式，然后再转向程序中剩下的部分。

第3步：构成新文件名，并对文件改名

作为最后一步，连接前一步生成的变量中的字符串，得到欧洲风格的日期：日期在月份之前。用下面的代码代替程序中最后3个TODO：

```
  #! python3
  # renameDates.py - Renames filenames with American MM-DD-YYYY date format
  # to European DD-MM-YYYY.

  --snip

--

      # Form the European-style filename.

❶     euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPar
```

```
                afterPart

        # Get the full, absolute file paths.

        absWorkingDir = os.path.abspath('.')

        amerFilename = os.path.join(absWorkingDir, amerFilename)

        euroFilename = os.path.join(absWorkingDir, euroFilename)

        # Rename the files.
```

```
❷     print('Renaming "%s" to "%s"...' % (amerFilename, euroFilename))

❸     #shutil.move(amerFilename, euroFilename) # uncomment after testing
```

将连接的字符串保存在名为euroFilename的变量中❶。然后将amerFilename中原来的文件名和新的euroFilename变量传递给shutil.move()函数，将该文件改名❸。

这个程序将shutil.move()调用注释掉，代之以打印出将被改名的文件名❷。先像这样运行程序，你可以确认文件改名是正确的。然后取消shutil.move()调用的注释，再次运行该程序，确实将这些文件改名。

第4步：类似程序的想法

有很多其他的理由，导致你需要对大量的文件改名。

- 为文件名添加前缀，诸如添加spam_，将eggs.txt改名为spam_eggs.txt。
- 将欧洲风格日期的文件改名为美国风格日期。
- 删除文件名中的0，诸如spam0042.txt。

9.5 项目：将一个文件夹备份到一个ZIP文件

假定你正在做一个项目，它的文件保存在C:\AlsPythonBook文件夹

中。你担心工作会丢失，所以希望为整个文件夹创建一个ZIP文件，作为“快照”。你希望保存不同的版本，希望ZIP文件的文件名每次创建时都有所变化。例如AlsPythonBook_1.zip、AlsPythonBook_2.zip、AlsPythonBook_3.zip，等等。你可以手工完成，但这有点烦人，而且可能不小心弄错ZIP文件的编号。运行一个程序来完成这个烦人的任务会简单得多。

针对这个项目，打开一个新的文件编辑器窗口，将它保存为backupToZip.py。

第1步：弄清楚ZIP文件的名称

这个程序的代码将放在一个名为backupToZip()的函数中。这样就更容易将该函数复制粘贴到其他需要这个功能的Python程序中。在这个程序的末尾，会调用这个函数进行备份。让你的程序看起来像这样：

```
  #! python3
  # backupToZip.py - Copies an entire folder and its contents into
  # a ZIP file whose filename increments.

❶ import zipfile, os

  def backupToZip(folder):
      # Backup the entire contents of "folder" into a ZIP file.

      folder = os.path.abspath(folder) # make sure folder is absolute

      # Figure out the filename this code should use based on
      # what files already exist.
❷     number = 1
❸     while True:
          zipFilename = os.path.basename(folder) + '_' + str(number) + '.zi
          if not os.path.exists(zipFilename):
              break
          number = number + 1

❹     # TODO: Create the ZIP file.

      # TODO: Walk the entire folder tree and compress the files in each fo
      print('Done.')

  backupToZip('C:\\delicious')
```

先完成基本任务：添加#!行，描述该程序做什么，并导入zipfile和os模块❶。

定义backupToZip()函数，它只接收一个参数，即folder。这个参数是一个字符串路径，指向需要备份的文件夹。该函数将决定它创建的ZIP文件使用什么文件名，然后创建该文件，遍历folder文件夹，将每个子文件夹和文件添加到ZIP文件中。在源代码中为这些步骤写下TODO注释，提醒你稍后来完成❹。

第一部分命名这个ZIP文件，使用folder的绝对路径的基本名称。如果要备份的文件夹是C:\delicious，ZIP文件的名称就应该是delicious_N.zip，第一次运行该程序时N=1，第二次运行时N=2，以此类推。

通过检查delicious_1.zip是否存在，然后检查delicious_2.zip是否存在，继续下去，可以确定N应该是什么。用一个名为number的变量表示N❷，在一个循环内不断增加它，并调用os.path.exists()来检查该文件是否存在❸。第一个不存在的文件名将导致循环break，因此它就发现了新ZIP文件的文件名。

第2步：创建新ZIP文件

接下来让我们创建ZIP文件。让你的程序看起来像这样：

```
 #! python3
 # backupToZip.py - Copies an entire folder and its contents into
 # a ZIP file whose filename increments.

 --snip

--
     while True:
         zipFilename = os.path.basename(folder) + '_' + str(number) + '.zi
         if not os.path.exists(zipFilename):
```

```
            break
        number = number + 1

    # Create the ZIP file.
    print('Creating %s...' % (zipFilename))
❶   backupZip = zipfile.ZipFile(zipFilename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each fo
    print('Done.')

backupToZip('C:\\delicious')
```

既然新ZIP文件的文件名保存在zipFilename变量中，你就可以调用zipfile.ZipFile()，实际创建这个ZIP文件❶。确保传入'w'作为第二个参数，这样ZIP文件以写模式打开。

第3步：遍历目录树并添加到ZIP文件

现在需要使用os.walk()函数，列出文件夹以及子文件夹中的每个文件。让你的程序看起来像这样：

```
  #! python3
  # backupToZip.py - Copies an entire folder and its contents into
  # a ZIP file whose filename increments.

  --snip

--

      # Walk the entire folder tree and compress the files in each folder.

❶     for foldername, subfolders, filenames in os.walk(folder):

         print('Adding files in %s...' % (foldername))

          # Add the current folder to the ZIP file.

❷         backupZip.write(foldername)

      # Add all the files in this folder to the ZIP file.
```

```
❸     for filename in filenames:

          newBase / os.path.basename(folder) + '_'

          if filename.startswith(newBase) and filename.endswith('.zip')

              continue # don't backup the backup ZIP files

          backupZip.write(os.path.join(foldername, filename))

      backupZip.close()

      print('Done.')
  backupToZip('C:\\delicious')
```

可以在for循环中使用os.walk()❶，在每次迭代中，它将返回这次迭代当前的文件夹名称、这个文件夹中的子文件夹，以及这个文件夹中的

文件名。

在这个for循环中，该文件夹被添加到ZIP文件❷。嵌套的for循环将遍历filenames列表中的每个文件❸。每个文件都被添加到ZIP文件中，以前生成的备份ZIP文件除外。

如果运行该程序，它产生的输出看起来像这样：

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

第二次运行它时，它将C:\delicious中的所有文件放进一个ZIP文件，命名为delicious_2.zip，以此类推。

第4步：类似程序的想法

你可以在其他程序中遍历一个目录树，将文件添加到压缩的ZIP归档文件中。例如，你可以编程做下面的事情：

- 遍历一个目录树，将特定扩展名的文件归档，诸如.txt或.py，并排除其他文件。
- 遍历一个目录树，将除.txt和.py文件以外的其他文件归档。
- 在一个目录树中查找文件夹，它包含的文件数最多，或者使用的磁盘空间最大。

9.6 小结

即使你是一个有经验的计算机用户，可能也会用鼠标和键盘手工处理文件。现在的文件浏览器使得处理少量文件的工作很容易。但有时候，如果用计算机的浏览器，你需要完成的任务可能要花几个小时。

os和shutil模块提供了一些函数，用于复制、移动、改名和删除文件。在删除文件时，你可能希望使用send2trash模块，将文件移动到回收站或垃圾箱，而不是永久地删除它们。在编程处理文件时，最好是先注释掉实际会复制/移动/改名/删除文件的代码，添加print()调用，这样你就可以运行该程序，验证它实际会做什么。

通常，你不仅需要对一个文件夹中的文件执行这些操作，而是对所有下级子文件夹执行操作。os.walk()函数将处理这个艰苦工作，遍历文件夹，这样你就可以专注于程序需要对其中的文件做什么。

zipfile模块提供了一种方法，用Python压缩和解压ZIP归档文件。和os和shutil模块中的文件处理函数一起使用，很容易将硬盘上任意位置的一些文件打包。和许多独立的文件相比，这些ZIP文件更容易上传到网站，或作为E-mail附件发送。

本书前面几章提供了源代码让你拷贝。但如果你编写自己的程序，可能在第一次编写时不会完美无缺。下一章将聚焦于一些Python模块，它们帮助你分析和调试程序，这样就能让程序很快正确运行。

9.7 习题

1．shutil.copy()和shutil.copytree()之间的区别是什么？

2．什么函数用于文件改名？

3．send2trash和shutil模块中的删除函数之间的区别是什么？

4．ZipFile对象有一个close()方法，就像File对象的close()方法。ZipFile对象的什么方法等价于File对象的open()方法？

9.8 实践项目

作为实践，编程完成下面的任务。

9.8.1 选择性拷贝

编写一个程序，遍历一个目录树，查找特定扩展名的文件（诸

如.pdf或.jpg）。不论这些文件的位置在哪里，将它们拷贝到一个新的文件夹中。

9.8.2 删除不需要的文件

一些不需要的、巨大的文件或文件夹占据了硬盘的空间，这并不少见。如果你试图释放计算机上的空间，那么删除不想要的巨大文件效果最好。但首先你必须找到它们。

编写一个程序，遍历一个目录树，查找特别大的文件或文件夹，比方说，超过100MB的文件（回忆一下，要获得文件的大小，可以使用 os模块的os.path.getsize()）。将这些文件的绝对路径打印到屏幕上。

9.8.3 消除缺失的编号

编写一个程序，在一个文件夹中，找到所有带指定前缀的文件，诸如spam001.txt, spam002.txt等，并定位缺失的编号（例如存在spam001.txt和spam003.txt，但不存在spam002.txt）。让该程序对所有后面的文件改名，消除缺失的编号。

作为附加的挑战，编写另一个程序，在一些连续编号的文件中，空出一些编号，以便加入新的文件。

第10章　调试

既然你已学习了足够的内容，可以编写更复杂的程序，可能就会在程序中发现不那么简单的缺陷。本章介绍了一些工具和技巧，用于寻找程序中缺陷的根源，帮助你更快更容易地修复缺陷。

程序员之间流传着一个老笑话：“编码占了编程工作量的90%，调试占了另外90%。”

计算机只会做你告诉它做的事情，它不会读懂你的心思，做你想要它做的事情。即使专业的程序员也一直在制造缺陷，所以如果你的程序有问题，不必感到沮丧。

好在，有一些工具和技巧可以确定你的代码在做什么，以及哪儿出了问题。首先，你要查看日志和断言。这两项功能可以帮助你尽早发现缺陷。一般来说，缺陷发现的越早，就越容易修复。

其次，你要学习如何使用调试器。调试器是IDLE的一项功能，它可以一次执行一条指令，在代码运行时，让你有机会检查变量的值，并追踪程序运行时值的变化。这比程序全速运行要慢得多，但可以帮助你查看程序运行时其中实际的值，而不是通过源代码推测值可能是什么。

10.1 抛出异常

当Python试图执行无效代码时，就会抛出异常。在第 3 章中，你已看到如何使用try和except语句来处理Python的异常，这样程序就可以从你预期的异常中恢复。但你也可以在代码中抛出自己的异常。抛出异常相当于是说：“停止运行这个函数中的代码，将程序执行转到except语句”。

抛出异常使用raise语句。在代码中，raise语句包含以下部分：

- raise关键字；
- 对Exception函数的调用；
- 传递给Exception函数的字符串，包含有用的出错信息。

例如，在交互式环境中输入以下代码：

```
>>> raise Exception('This is the error message.')




Traceback (most recent call last):
   File "", line 1, in
     raise Exception('This is the error message.')
Exception: This is the error message.
```

如果没有try和except语句覆盖抛出异常的raise语句，该程序就会崩溃，并显示异常的出错信息。

通常是调用该函数的代码知道如何处理异常，而不是该函数本身。所以你常常会看到raise语句在一个函数中，try和except语句在调用该函数的代码中。例如，打开一个新的文件编辑器窗口，输入以下代码，并保存为boxPrint.py：

```
  def boxPrint(symbol, width, height):
      if len(symbol) != 1:
❶         raise Exception('Symbol must be a single character string.')
      if width <= 2:
❷         raise Exception('Width must be greater than 2.')
      if height <= 2:
❸         raise Exception('Height must be greater than 2.')
      print(symbol * width)
      for i in range(height - 2):
          print(symbol + (' ' * (width - 2)) + symbol)
      print(symbol * width)

  for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
      try:
          boxPrint(sym, w, h)
❹     except Exception as err:
❺         print('An exception happened: ' + str(err))
```

这里我们定义了一个boxPrint() 函数，它接受一个字符、一个宽度和一个高度。它按照指定的宽度和高度，用该字符创建了一个小盒子的图像。这个盒子被打印到屏幕上。

假定我们希望该字符是一个字符，宽度和高度要大于2。我们添加了if语句，如果这些条件没有满足，就抛出异常。稍后，当我们用不同的参数调用boxPrint()时，try/except语句就会处理无效的参数。

这个程序使用了except语句的except Exception as err形式❹。如果boxPrint()返回一个Exception对象❶❷❸，这条语句就会将它保存在名为err的变量中。Exception对象可以传递给str()，将它转换为一个字符串，得到用户友好的出错信息❺。运行boxPrint.py，输出看起来像这样：

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

使用try和except语句，你可以更优雅地处理错误，而不是让整个程序崩溃。

10.2 取得反向跟踪的字符串

如果Python遇到错误，它就会生成一些错误信息，称为“反向跟踪”。反向跟踪包含了出错消息、导致该错误的代码行号，以及导致该

错误的函数调用的序列。这个序列称为“调用栈”。

在IDLE中打开一个新的文件编辑器窗口，输入以下程序，并保存为error Example.py：

```
def spam():
     bacon()
def bacon():
     raise Exception('This is the error message.')

spam()
```

如果运行errorExample.py，输出看起来像这样：

```
Traceback (most recent call last):
   File "errorExample.py", line 7, in <module>
     spam()
   File "errorExample.py", line 2, in spam
     bacon()
   File "errorExample.py", line 5, in bacon
     raise Exception('This is the error message.')
Exception: This is the error message.
```

通过反向跟踪，可以看到该错误发生在第5行，在bacon() 函数中。这次特定的bacon() 调用来自第2行，在spam() 函数中，它又在第7行被调用的。在从多个位置调用函数的程序中，调用栈就能帮助你确定哪次调用导致了错误。

只要抛出的异常没有被处理，Python 就会显示反向跟踪。但你也可以调用traceback.format_exc()，得到它的字符串形式。如果你希望得到异常的反向跟踪的信息，但也希望except语句优雅地处理该异常，这个函数就很有用。在调用该函数之前，需要导入Python的traceback模块。

例如，不是让程序在异常发生时就崩溃，可以将反向跟踪信息写入

一个日志文件，并让程序继续运行。稍后，在准备调试程序时，可以检查该日志文件。在交互式环境中输入以下代码：

```
>>> import traceback

>>> try:

          raise Exception('This is the error message.')

except:

          errorFile = open('errorInfo.txt', 'w')

          errorFile.write(traceback.format_exc())

          errorFile.close()

          print('The traceback info was written to errorInfo.txt.')
```

```
116
The traceback info was written to errorInfo.txt.
```

write() 方法的返回值是116，因为116个字符被写入到文件中。反向跟踪文本被写入errorInfo.txt。

```
Traceback (most recent call last):
   File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

10.3 断言

“断言”是一个心智正常的检查，确保代码没有做什么明显错误的事情。这些心智正常的检查由assert语句执行。如果检查失败，就会抛出异常。在代码中，assert语句包含以下部分：

- assert关键字；
- 条件（即求值为True或False的表达式）；
- 逗号；
- 当条件为False时显示的字符串。

例如，在交互式环境中输入以下代码：

```
>>> podBayDoorStatus = 'open'

>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open"

>>> podBayDoorStatus = 'I\'m sorry, Dave. I\'m afraid I can't do that.''

>>> assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open"

Traceback (most recent call last):
   File "< pyshell#10>", line 1, in < module>
     assert podBayDoorStatus == 'open', 'The pod bay doors need to be "open
AssertionError: The pod bay doors need to be "open".
```

这里将podBayDoorStatus设置为 'open'，所以从此以后，我们充分期望这个变量的值是 'open'。在使用这个变量的程序中，基于这个值是 'open' 的假定，我们可能写下了大量的代码，即这些代码依赖于它是 'open'，才能按照期望工作。所以添加了一个断言，确保假定podBayDoorStatus是 'open' 是对的。这里，我们加入了信息 'The pod bay doors need to be "open".'，这样如果断言失败，就很容易看到哪里出了错。

稍后，假如我们犯了一个明显的错误，把另外的值赋给podBayDoorStatus，但在很多行代码中，我们并没有意识到这一点。这

个断言会抓住这个错误，清楚地告诉我们出了什么错。

在日常英语中，assert语句是说："我断言这个条件为真，如果不为真，程序中什么地方就有一个缺陷。"不像异常，代码不应该用try和except处理assert语句。如果assert失败，程序就应该崩溃。通过这样的快速失败，产生缺陷和你第一次注意到该缺陷之间的时间就缩短了。这将减少为了寻找导致该缺陷的代码，而需要检查的代码量。

断言针对的是程序员的错误，而不是用户的错误。对于那些可以恢复的错误（诸如文件没有找到，或用户输入了无效的数据），请抛出异常，而不是用assert语句检测它。

10.3.1 在交通灯模拟中使用断言

假定你在编写一个交通信号灯的模拟程序。代表路口信号灯的数据结构是一个字典，以 'ns' 和 'ew' 为键，分别表示南北向和东西向的信号灯。这些键的值可以是 'green'、'yellow' 或 'red' 之一。代码看起来可能像这样：

```
market_2nd = {'ns': 'green', 'ew': 'red'}
mission_16th = {'ns': 'red', 'ew': 'green'}
```

这两个变量将针对Market街和第2街路口，以及Mission街和第16街路口。作为项目启动，你希望编写一个switchLights() 函数，它接受一个路口字典作为参数，并切换红绿灯。

开始你可能认为，switchLights() 只要将每一种灯按顺序切换到下一种颜色：'green' 值应该切换到 'yellow'，'yellow' 应该切换到 'red'，'red' 应该切换到'green'。实现这个思想的代码看起来像这样：

```
def switchLights(stoplight):
     for key in stoplight.keys():
          if stoplight[key] == 'green':
               stoplight[key] = 'yellow'
          elif stoplight[key] == 'yellow':
               stoplight[key] = 'red'
```

```
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'

switchLights(market_2nd)
```

你可能已经发现了这段代码的问题，但假设你编写了剩下的模拟代码，有几千行，但没有注意到这个问题。当最后运行时，程序没有崩溃，但虚拟的汽车撞车了！

因为你已经编写了剩下的程序，所以不知道缺陷在哪里。也许在模拟汽车的代码中，或者在模拟司机的代码中。可能需要花几个小时追踪缺陷，才能找到switchLights() 函数。

但如果在编写switchLights() 时，你添加了断言，确保至少一个交通灯是红色，可能在函数的底部添加这样的代码：

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stopligh
```

有了这个断言，程序就会崩溃，并提供这样的出错信息：

```
  Traceback (most recent call last):
    File "carSim.py", line 14, in <module>
      switchLights(market_2nd)
    File "carSim.py", line 13, in switchLights
      assert 'red' in stoplight.values(), 'Neither light is red! ' + str(s
❶ AssertionError: Neither light is red! {'ns': 'yellow', 'ew': 'green'}
```

这里重要的一行是AssertionError❶。虽然程序崩溃并非如你所愿，但它马上指出了心智正常检查失败：两个方向都没有红灯，这意味着两个方向的车都可以走。在程序执行中尽早快速失败，可以省去将来大量

的调试工作。

10.3.2 禁用断言

在运行Python时传入-O选项，可以禁用断言。如果你已完成了程序的编写和测试，不希望执行心智正常检测，从而减慢程序的速度，这样就很好（尽管大多数断言语句所花的时间，不会让你觉察到速度的差异）。断言是针对开发的，不是针对最终产品。当你将程序交给其他人运行时，它应该没有缺陷，不需要进行心智正常检查。如何用-O选项启动也许并不疯狂的程序，详细内容请参考附录B。

10.4 日志

如果你曾经在代码中加入print() 语句，在程序运行时输出某些变量的值，你就使用了记日志的方式来调试代码。记日志是一种很好的方式，可以理解程序中发生的事，以及事情发生的顺序。Python的logging模块使得你很容易创建自定义的消息记录。这些日志消息将描述程序执行何时到达日志函数调用，并列出你指定的任何变量当时的值。另一方面，缺失日志信息表明有一部分代码被跳过，从未执行。

10.4.1 使用日志模块

要启用logging模块，在程序运行时将日志信息显示在屏幕上，请将下面的代码复制到程序顶部（但在Python的#!行之下）：

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname
-   %(message)s')
```

你不需要过于担心它的工作原理，但基本上，当 Python 记录一个事件的日志时，它会创建一个LogRecord对象，保存关于该事件的信息。logging模块的函数让你指定想看到的这个LogRecord对象的细节，以及希望的细节展示方式。

假如你编写了一个函数，计算一个数的阶乘。在数学上，4 的阶乘是1 × 2 × 3 × 4，即24。7的阶乘是1 × 2 × 3 × 4 × 5 × 6 × 7，即5040。打开一个新的文件编辑器窗口，输入以下代码。其中有一个缺陷，但你也会输入一些日志信息，帮助你弄清楚哪里出了问题。将该程序保存为factorialLog.py。

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname
-   %(message)s')
logging.debug('Start of program')

def factorial(n):
     logging.debug('Start of factorial(%s%%)' % (n))
     total = 1
     for i in range(n + 1):
           total *= i
           logging.debug('i is ' + str(i) + ', total is ' + str(total))
     logging.debug('End of factorial(%s%%)' % (n))
     return total

print(factorial(5))
logging.debug('End of program')
```

这里，我们在想打印日志信息时，使用logging.debug() 函数。这个debug() 函数将调用basicConfig()，打印一行信息。这行信息的格式是我们在 basicConfig()函数中指定的，并且包括我们传递给 debug() 的消息。print（factorial（5））调用是原来程序的一部分，所以就算禁用日志信息，结果仍会显示。

这个程序的输出就像这样：

```
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
```

```
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

factorial() 函数返回0作为5的阶乘，这是不对的。for循环应该用从1到5的数，乘以total的值。但logging.debug() 显示的日志信息表明，i变量从0开始，而不是1。因为0乘任何数都是0，所以接下来的迭代中，total的值都是错的。日志消息提供了可以追踪的痕迹，帮助你弄清楚何时事情开始不对。

将代码行for i in range（n + 1）：改为for i in range（1，n + 1）：，再次运行程序。输出看起来像这样：

```
2015-05-23 17:13:40,650 - DEBUG - Start of program
2015-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2015-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2015-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2015-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2015-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2015-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2015-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2015-05-23 17:13:40,666 - DEBUG - End of program
```

factorial（5）调用正确地返回120。日志消息表明循环内发生了什么，这直接指向了缺陷。

你可以看到，logging.debug() 调用不仅打印出了传递给它的字符串，而且包含一个时间戳和单词DEBUG。

10.4.2 不要用**print()**调试

输入import logging和logging.basicConfig（level=logging.DEBUG, format='% (asctime)s - %(levelname)s - %(message)s'）有一点不方便。你

可能想使用print() 调用代替，但不要屈服于这种诱惑！在调试完成后，你需要花很多时间，从代码中清除每条日志消息的print() 调用。你甚至有可能不小心删除一些print() 调用，而它们不是用来产生日志消息的。日志消息的好处在于，你可以随心所欲地在程序中想加多少就加多少，稍后只要加入一次logging.disable（logging.CRITICAL）调用，就可以禁止日志。不像print()，logging模块使得显示和隐藏日志信息之间的切换变得很容易。

日志消息是给程序员的，不是给用户的。用户不会因为你便于调试，而想看到的字典值的内容。请将日志信息用于类似这样的目的。对于用户希望看到的消息，例如“文件未找到”或者“无效的输入，请输入一个数字”，应该使用print() 调用。我们不希望禁用日志消息之后，让用户看不到有用的信息。

10.4.3 日志级别

“日志级别”提供了一种方式，按重要性对日志消息进行分类。5个日志级别如表10-1所示，从最不重要到最重要。利用不同的日志函数，消息可以按某个级别记入日志。

表**10-1 Python**中的日志级别

级别	日志函数	描述
DEBUG	logging.debug()	最低级别。用于小细节。通常只有在诊断问题时，你才会关心这些消息
INFO	logging.info()	用于记录程序中一般事件的信息，或确认一切工作正常
WARNING	logging.warning()	用于表示可能的问题，它不会阻止程序的工作，但将来可能会
ERROR	logging.error()	用于记录错误，它导致程序做某事失败
		最高级别。用于表示致命的错误，它导致或将要导致

CRITICAL	logging.critical()	程序完全停止工作

日志消息作为一个字符串，传递给这些函数。日志级别是一种建议。归根到底，还是由你来决定日志消息属于哪一种类型。在交互式环境中输入以下代码：

```
>>> import logging

>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -

%(levelname)s - %(message)s')

>>> logging.debug('Some debugging details.')

2015-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('The logging module is working.')

2015-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('An error message is about to be logged.')
```

```
2015-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2015-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover!')
2015-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

日志级别的好处在于，你可以改变想看到的日志消息的优先级。向basicConfig()函数传入logging.DEBUG作为level关键字参数，这将显示所有日志级别的消息（DEBUG是最低的级别）。但在开发了更多的程序后，你可能只对错误感兴趣。在这种情况下，可以将basicConfig() 的level参数设置为logging.ERROR，这将只显示ERROR和CRITICAL消息，跳过DEBUG、INFO和WARNING消息。

10.4.4 禁用日志

在调试完程序后，你可能不希望所有这些日志消息出现在屏幕上。logging. disable() 函数禁用了这些消息，这样就不必进入到程序中，手工删除所有的日志调用。只要向logging.disable() 传入一个日志级别，它就会禁止该级别和更低级别的所有日志消息。所以，如果想要禁用所有日志，只要在程序中添加logging. disable（logging.CRITICAL）。例如，在交互式环境中输入以下代码：

```
>>> import logging
```

```
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s - %(message)s')
>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

因为logging.disable() 将禁用它之后的所有消息，你可能希望将它添加到程序中接近import logging代码行的位置。这样就很容易找到它，根

据需要注释掉它，或取消注释，从而启用或禁用日志消息。

10.4.5 将日志记录到文件

除了将日志消息显示在屏幕上，还可以将它们写入文本文件。logging.basic Config() 函数接受filename关键字参数，像这样：

```
import logging
logging.basicConfig(filename='myProgramLog.txt'





, level=logging.DEBUG, format='
%(asctime)s - %(levelname)s - %(message)s')
```

日志信息将被保存到myProgramLog.txt文件中。虽然日志消息很有用，但它们可能塞满屏幕，让你很难读到程序的输出。将日志信息写入到文件，让屏幕保持干净，又能保存信息，这样在运行程序后，可以阅读这些信息。可以用任何文件编辑器打开这个文本文件，诸如Notepad或TextEdit。

10.5 IDLE的调试器

“调试器”是IDLE的一项功能，让你每次执行一行程序。调试器将运行一行代码，然后等待你告诉它继续。像这样让程序运行“在调试器之下”，你可以随便花多少时间，检查程序运行时任意一个时刻的变量的值。对于追踪缺陷，这是一个很有价值的工具。

要启用IDLE的调试器，就在交互式环境窗口中点击DebugDebugger。这将打开调试控制（Debug Control）窗口，如图10-1所示。

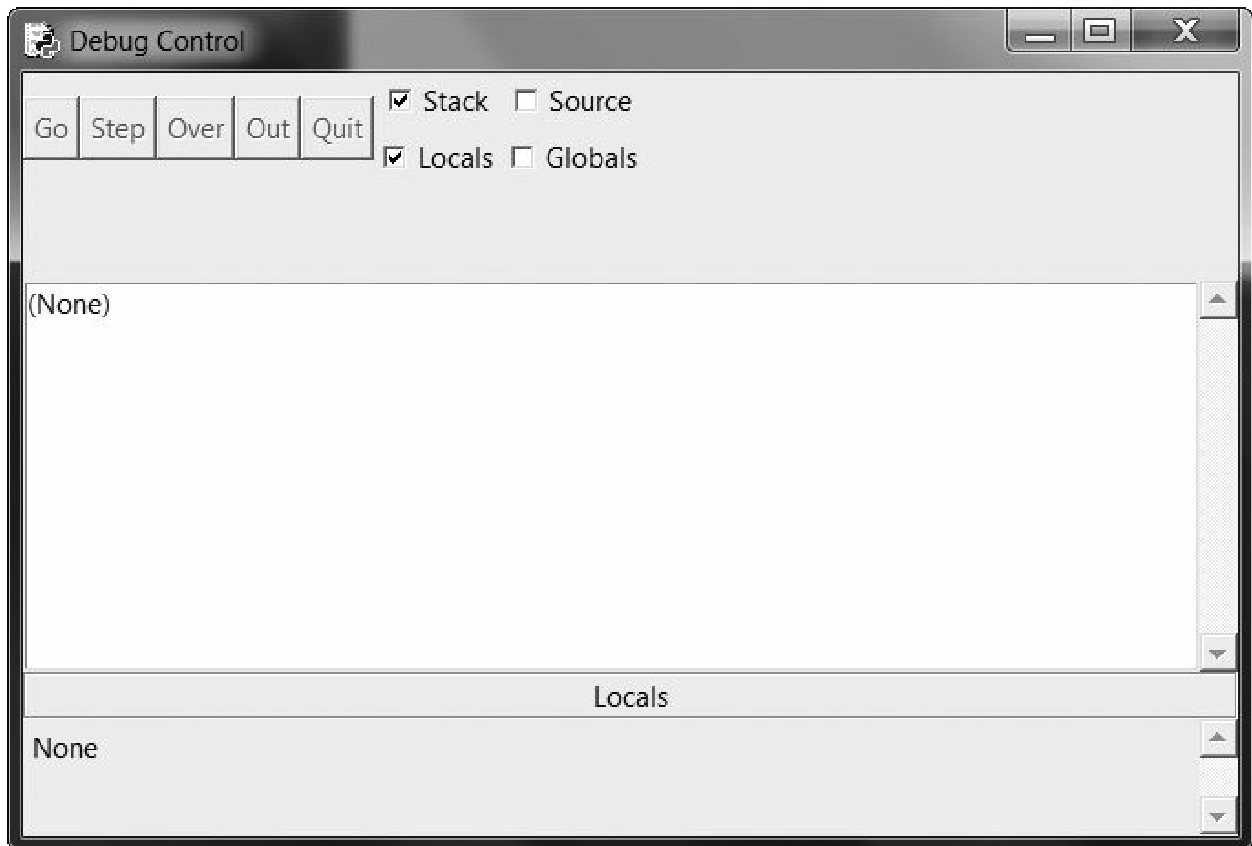


图10-1 调试控制窗口

当调试控制窗口出现后，勾选全部4个复选框：Stack、Locals、Source和Globals。这样窗口将显示全部的调试信息。调试控制窗口显示时，只要你从文件编辑器运行程序，调试器就会在第一条指令之前暂停执行，并显示下面的信息：

- 将要执行的代码行；
- 所有局部变量及其值的列表；
- 所有全局变量及其值的列表。

你会注意到，在全局变量列表中，有一些变量你没有定义，诸如__builtins__、__doc__、__file__，等等。它们是Python在运行程序时，自动设置的变量。这些变量的含义超出了本书的范围，你可以暂时忽略它们。

程序将保持暂停，直到你按下调试控制窗口的5个按钮中的一个：Go、Step、Over、Out或Quit。

10.5.1 Go

点击Go按钮将导致程序正常执行至终止，或到达一个“断点”（断点在本章稍后介绍）。如果你完成了调试，希望程序正常继续，就点击Go按钮。

10.5.2 Step

点击Step按钮将导致调试器执行下一行代码，然后再次暂停。如果变量的值发生了变化，调试控制窗口的全局变量和局部变量列表就会更新。如果下一行代码是一个函数调用，调试器就会“步入”那个函数，跳到该函数的第一行代码。

10.5.3 Over

点击Over按扭将执行下一行代码，与Step按钮类似。但是，如果下一行代码是函数调用，Over按钮将“跨过”该函数的代码。该函数的代码将以全速执行，调试器将在该函数返回后暂停。例如，如果下一行代码是print() 调用，你实际上不关心内建print() 函数中的代码，只希望传递给它的字符串打印在屏幕上。出于这个原因，使用Over按钮比使用Step按钮更常见。

10.5.4 Out

点击Out按钮将导致调试器全速执行代码行，直到它从当前函数返回。如果你用Step按钮进入了一个函数，现在只想继续执行指令，直到该函数返回，那就点击Out按钮，从当前的函数调用“走出来”。

10.5.5 Quit

如果你希望完全停止调试，不必继续执行剩下的程序，就点击Quit按钮。Quite按钮将马上终止该程序。如果你希望再次正常运行你的程序，就再次选择DebugDebugger，禁用调试器。

10.5.6 调试一个数字相加的程序

打开一个新的文件编辑器窗口，输入以下代码：

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

将它保存为buggyAddingProgram.py，不启用调试器，第一次运行它。程序的输出像这样：

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

这个程序没有崩溃，但求和显然是错的。让我们启用调试控制窗口，再次运行它，这次在调试器控制之下。

当你按下F5或选择RunRun Module（启用DebugDebugger，选中调试控制窗口的所有4个复选框），程序启动时将暂停在第1行。调试器总是会暂停在它将要执行的代码行上。调试控制窗口看起来如图10-2所示。

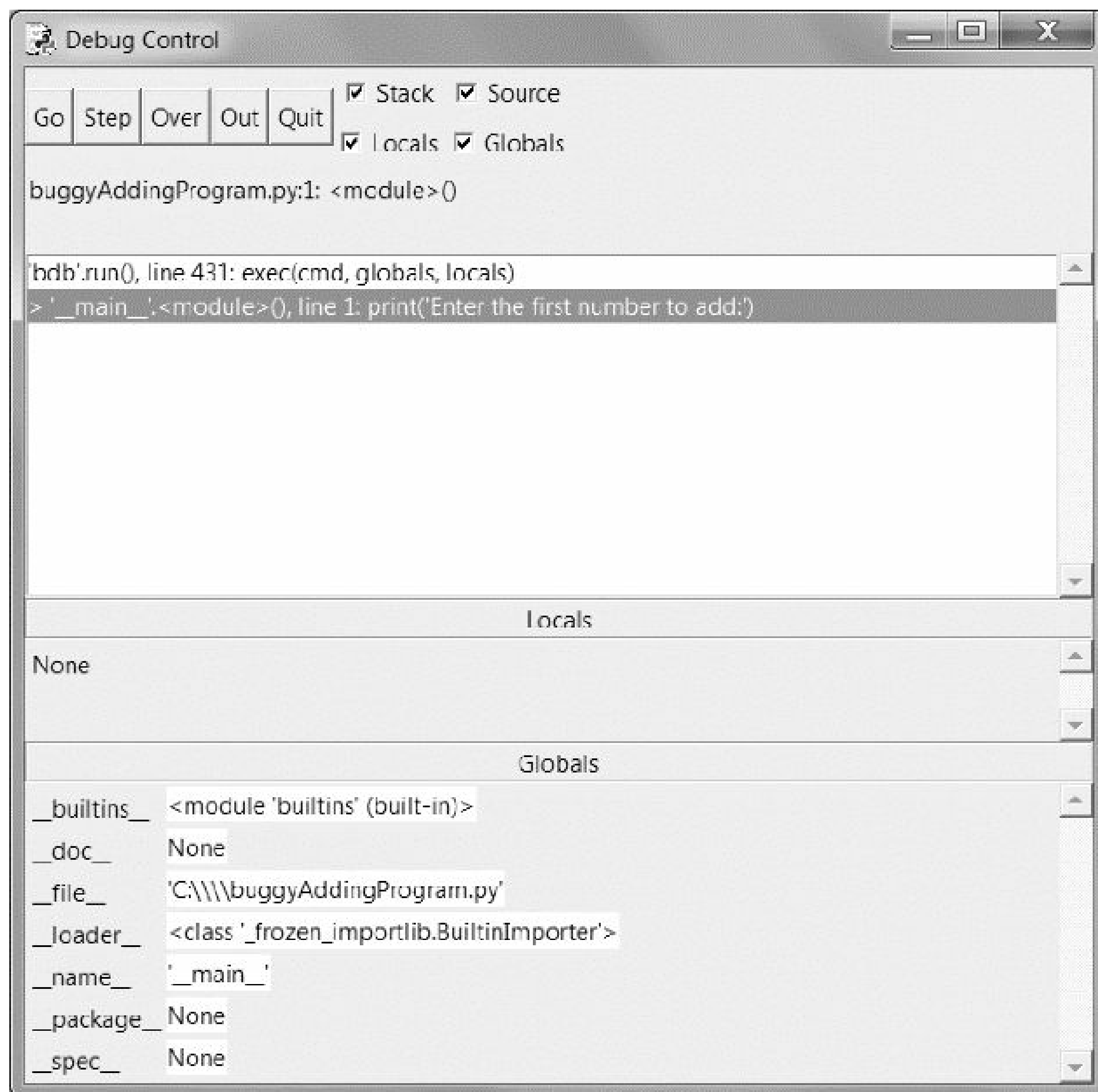


图10-2 程序第一次在调试器下运行时的调试控制窗口

点击一次Over按钮，执行第一个print() 调用。这里应该使用Over按钮，而不是Step，因为你不希望进入到print() 函数的代码中。调试控制窗口将更新到第2行，文件编辑器窗口的第2行将高亮显示，如图10-3所

示。这告诉你程序当前执行到 哪里。

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

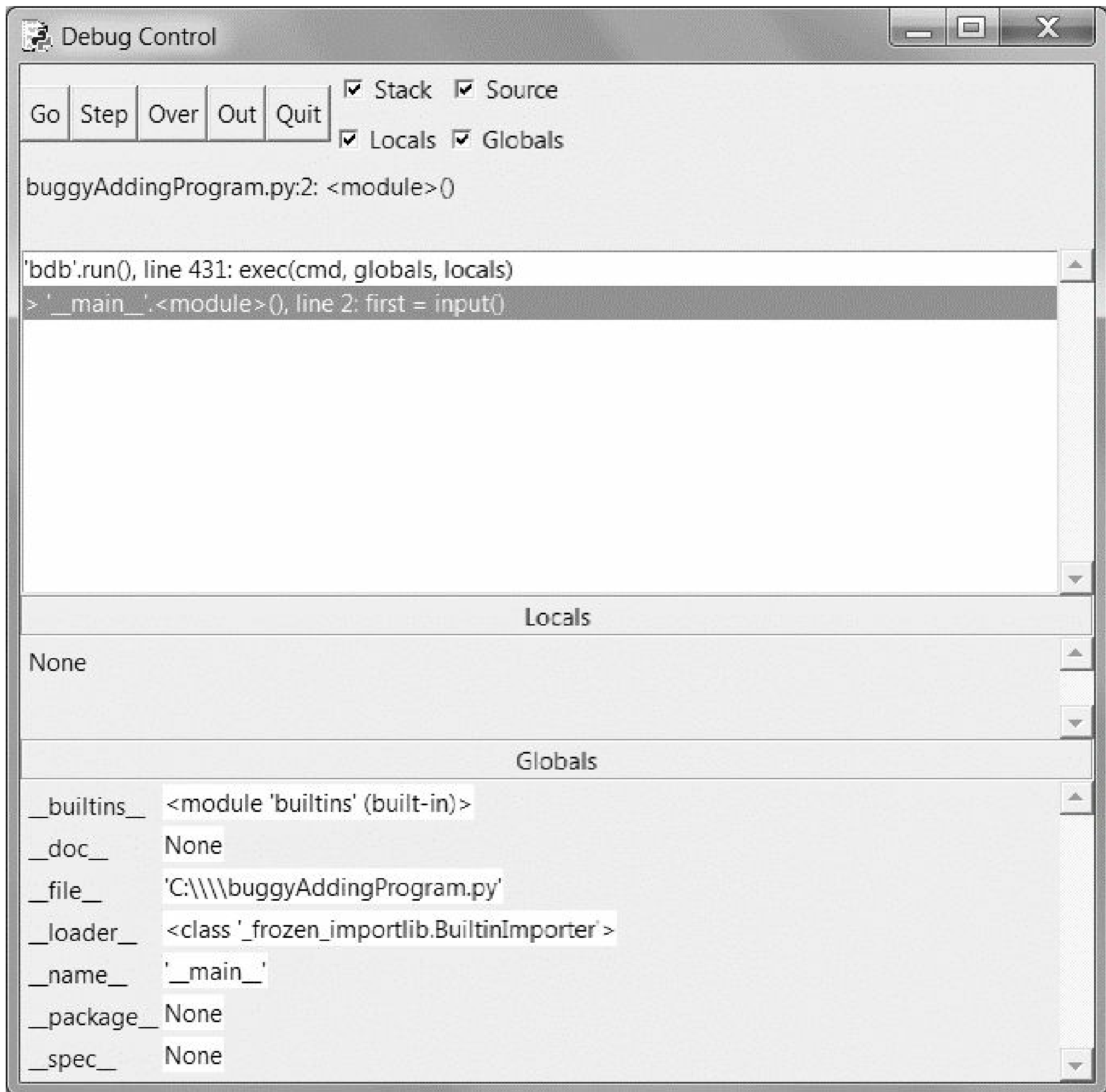


图10-3 点击Over按钮后的调试控制窗口

再次点击Over按钮，执行input() 函数调用，当IDLE等待你在交互式环境窗口中为input() 调用输入内容时，调试控制窗口中的按钮将被禁用。输入5并按回车。调试控制窗口按钮将重新启用。

继续点击Over按钮，输入3和42作为接下来的两个数，直到调试器位于第7行，程序中最后的print() 调用。调试控制窗口应该如图10-4所示。可以看到，在全局变量的部分，第一个、第二个和第三个变量设置为字符串值，而不是整型值。当最后一行执行时，这些字符串连接起

来，而不是加起来，导致了这个缺陷。

用调试器单步执行程序很有用，但也可能很慢。你常常希望程序正常运行，直到它到达特定的代码行。你可以使用断点，让调试器做到这一点。

10.5.7 断点

“断点”可以设置在特定的代码行上，当程序执行到达该行时，它迫使调试器暂停。在一个新的文件编辑器窗口中，输入以下程序，它模拟投掷1000次硬币。将它保存为coinFlip.py。

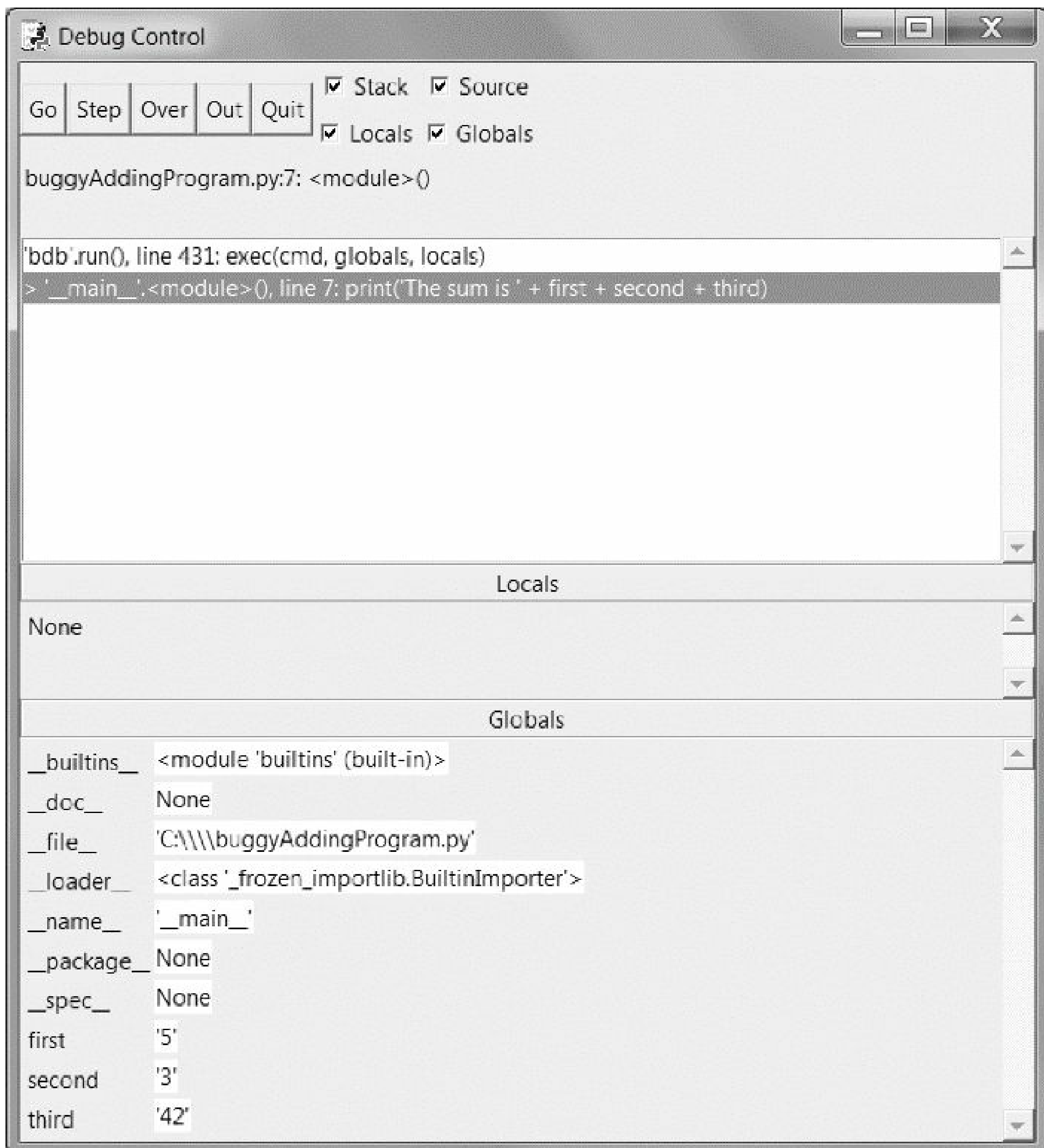


图10-4 在最后一行的调试控制窗口。这些变量被设置为字符串，导致了这个缺陷

```
  import random
  heads = 0
  for i in range(1, 1001):
❶     if random.randint(0, 1) == 1:
          heads = heads + 1
      if i == 500:
```

```
❷            print('Halfway done!')
  print('Heads came up ' + str(heads) + ' times.')
```

在半数时间里，random.randint（0，1）调用❶将返回0，在另外半数时间将返回1。这可以用来模拟50/50的硬币投掷，其中1代表正面。当不用调试器运行该程序时，它很快输出下面的内容：

```
Halfway done!
Heads came up 490 times.
```

如果启用调试器运行这个程序，就必须点击几千次Over按钮，程序才能结束。如果你对程序执行到一半时heads的值感兴趣，等1000次硬币投掷完500次，可以在代码行print（'Halfway done!'）❷上设置断点。要设置断点，在文件编辑器中该行代码上点击右键，并选择Set Breakpoint，如图10-5所示。

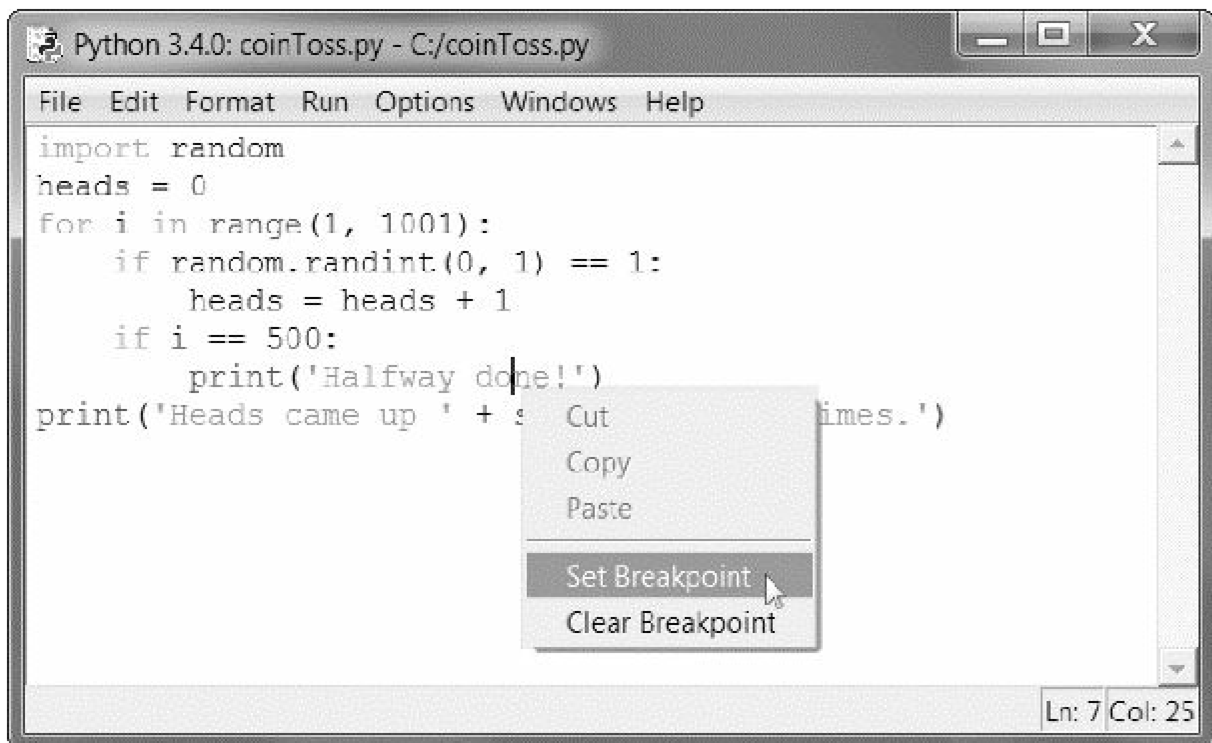


图10-5 设置断点

你不会在if语句上设置断点，因为if语句会在循环的每次迭代中都执行。通过在if语句内的代码上设置断点，调试器就会只在执行进入if语句时才中断。

带有断点的代码行会在文件编辑器中以黄色高亮显示。如果在调试器下运行该程序，开始它会暂停在第一行，像平时一样。但如果点击Go，程序将全速运行，直到设置了断点的代码行。然后可以点击Go、Over、Step或Out，正常继续。

如果希望清除断点，在文件编辑器中该行代码上点击右键，并从菜单中选择Clear Breakpoint。黄色高亮消失，以后调试器将不会在该行代码上中断。

10.6 小结

断言、异常、日志和调试器，都是在程序中发现和预防缺陷的有用工具。用Python语句实现的断言，是实现心智正常检查的好方式。如果必要的条件没有保持为True，它将尽早给出警告。断言所针对的错误，

是程序不应该尝试恢复的，而是应该快速失败。否则，你应该抛出异常。

异常可以由try和except语句捕捉和处理。logging模块是一种很好的方式，可以在运行时查看代码的内部，它比使用print() 函数要方便得多，因为它有不同的日志级别，并能将日志写入文本文件。

调试器让你每次单步执行一行代码。或者，可以用正常速度运行程序，并让调试器暂停在设置了断点的代码行上。利用调试器，你可以看到程序在运行期间，任何时候所有变量的值。

这些调试工具和技术将帮助你编写正确工作的程序。不小心在代码中引入缺陷，这是不可避免的，不论你有多少年的编码经验。

10.7 习题

1．写一条 assert 语句，如果变量 spam 是一个小于 10 的整数，就触发AssertionError。

2．编写一条assert语句，如果eggs和bacon包含的字符串彼此相同，而且不论大小写如何，就触发AssertionError（也就是说，'hello' 和 'hello' 被认为相同，'goodbye' 和 'GOODbye' 也被认为相同）。

3．编写一条assert语句，总是触发AssertionError。

4．为了能调用logging.debug()，程序中必须加入哪两行代码？

5．为了让logging.debug() 将日志消息发送到名为programLog.txt的文件中，程序必须加入哪两行代码？

6．5个日志级别是什么？

7．你可以加入哪一行代码，禁用程序中所有的日志消息？

8．显示同样的消息，为什么使用日志消息比使用print() 要好？

9．调试控制窗口中的Step、Over和Out按钮有什么区别？

10．在点击调试控制窗口中的Go按钮后，调试器何时会停下来？

11．什么是断点？

12．在IDLE中，如何在一行代码上设置断点？

10.8 实践项目

作为实践，编程完成下面的任务。

调试硬币抛掷

下面程序的意图是一个简单的硬币抛掷猜测游戏。玩家有两次猜测机会（这是一个简单的游戏）。但是，程序中有一些缺陷。让程序运行几次，找出缺陷，使该程序能正确运行。

```
import random
guess = ''
while guess not in ('heads', 'tails'):
     print('Guess the coin toss! Enter heads or tails:')
     guess = input()
toss = random.randint(0, 1) # 0 is tails, 1 is heads
if toss == guess:
     print('You got it!')
else:
     print('Nope! Guess again!')
     guesss = input()
     if toss == guess:
          print('You got it!')
     else:
          print('Nope. You are really bad at this game.')
```

第11章　从Web抓取信息

少数可怕的时候，我没有 Wi-Fi。这时才意识到，我在计算机上所做的事，有多少实际上是在因特网上做的事。完全出于习惯，我会发现自己尝试收邮件、阅读朋友的推特，或回答问题：“在Kurtwood Smith演出1987年的机械战警之前，曾经演过主角吗？”[1]

因为计算机上如此多的工作都与因特网有关，所以如果程序能上网就太好了。“Web 抓取”是一个术语，即利用程序下载并处理来自Web的内容。例如，Google运行了许多web抓取程序，对网页进行索引，实现它的搜索引擎。在本章中，你将学习几个模块，让在Python中抓取网页变得很容易。

webbrowser：是Python自带的，打开浏览器获取指定页面。

requests：从因特网上下载文件和网页。

Beautiful Soup：解析HTML，即网页编写的格式。

selenium：启动并控制一个Web浏览器。selenium能够填写表单，并模拟鼠标在这个浏览器中点击。

11.1 项目：利用webbrowser模块的mapIt.py

webbrowser模块的open()函数可以启动一个新浏览器，打开指定的URL。在交互式环境中输入以下代码：

```
>>> import webbrowser



>>> webbrowser.open('http://inventwithpython.com/')
```

Web浏览器的选项卡将打开URL http://inventwithpython.com/ 。这大概就是webbrowser模块能做的唯一的事情。既使如此，open()函数确实让一些有趣的事情成为可能。例如，将一条街道的地址拷贝到剪贴板，并在Google地图上打开它的地图，这是很繁琐的事。你可以让这个任务减少几步，写一个简单的脚本，利用剪贴板中的内容在浏览器中自动加载地图。这样，你只要将地址拷贝到剪贴板，运行该脚本，地图就会加载。

你的程序需要做到：

- 从命令行参数或剪贴板中取得街道地址。
- 打开Web浏览器，指向该地址的Google地图页面。

这意味着代码需要做下列事情：

- 从sys.argv读取命令行参数。
- 读取剪贴板内容。
- 调用webbrowser.open()函数打开外部浏览器。

打开一个新的文件编辑器窗口，将它保存为mapIt.py。

第**1**步：弄清楚**URL**

根据附录B中的指导，建立mapIt.py，这样当你从命令行运行它时，例如

```
C:\> mapit 870 Valencia St, San Francisco, CA 94110
```

该脚本将使用命令行参数，而不是剪贴板。如果没有命令行参数，程序就知道要使用剪贴板的内容。

首先你需要弄清楚，对于指定的街道地址，要使用怎样的URL。你在浏览器中打开http://maps.google.com/ 并查找一个地址时，地址栏中的URL看起来就像这样：https://www.google.com/maps/place/870+Valencia+St/@37.7590311,-122.4215096,17z/data=!3m1!4b1!4m2!3m1!1s0x808f7e3dadc07a37:0xc86b0b2bb93b73d8.

地址就在URL中，但其中还有许多附加的文本。网站常常在URL中添加额外的数据，帮助追踪访问者或定制网站。但如果你尝试使用https://www.google.com/maps/place/870+Valencia+St+San+Francisco+CA/，会发现仍然可以到达正确的页面。所以你的程序可以设置为打开一个浏览器，访问'https://www.google.com/ maps/place/your_address_string'（其中your_address_string是想查看地图的地址）。

第**2**步：处理命令行参数

让你的代码看起来像这样：

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])

# TODO: Get address from clipboard.
```

在程序的#!行之后，需要导入webbrowser模块，用于加载浏览器；导入sys模块，用于读入可能的命令行参数。sys.argv变量保存了程序的文件名和命令行参数的列表。如果这个列表中不只有文件名，那么

len(sys.argv)的返回值就会大于1，这意味着确实提供了命令行参数。

命令行参数通常用空格分隔，但在这个例子中，你希望将所有参数解释为一个字符串。因为sys.argv是字符串的列表，所以你可以将它传递给join()方法，这将返回一个字符串。你不希望程序的名称出现在这个字符串中，所以不是使用sys.argv，而是使用sys.argv[1:]，砍掉这个数组的第一个元素。这个表达式求值得到的字符串，保存在address变量中。

如果运行程序时在命令行中输入以下内容：

```
mapit 870 Valencia St, San Francisco, CA 94110
```

…sys.argv变量将包含这样的列表值：

```
 ['mapIt.py', '870', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110
```

address变量将包含字符串'870 Valencia St, San Francisco, CA 94110'。

第**3**步：处理剪贴板内容，加载浏览器

让你的代码看起来像这样：

```
#! python3
# mapIt.py - Launches a map in the browser using an address from the
# command line or clipboard.

import webbrowser, sys, pyperclip
```

```
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

webbrowser.open('https://www.google.com/maps/place/' + address)
```

如果没有命令行参数，程序将假定地址保存在剪贴板中。可以用pyperclip.paste()取得剪贴板的内容，并将它保存在名为address的变量中。最后，启动外部浏览器访问Google地图的URL，调用webbrowser.open()。

虽然你写的某些程序将完成大型任务，为你节省数小时的时间，但使用一个程序，在每次执行一个常用任务时节省几秒钟时间，比如取得一个地址的地图，这同样令人满意。表11-1比较了有mapIt.py和没有它时，显示地图所需的步骤。

表**11-1** 不用和利用**mapIt.py**取得地图

手工取得地图	利用**mapIt.py**
高亮标记地址	高亮标记地址
拷贝地址	拷贝地址
打开Web浏览器	运行mapIt.py
打开*http://maps.google.com/*	
点击地址文本字段	
拷贝地址	
按回车	

看到程序让这个任务变得不那么繁琐了吗？

第**4**步：类似程序的想法

只要你有一个URL，webbrowser模块就让用户不必打开浏览器，而直接加载一个网站。其他程序可以利用这项功能完成以下任务：

- 在独立的浏览器标签中，打开一个页面中的所有链接。
- 用浏览器打开本地天气的URL。
- 打开你经常查看的几个社交网站。

11.2 用**requests**模块从**Web**下载文件

requests模块让你很容易从Web下载文件，不必担心一些复杂的问

题，诸如网络错误、连接问题和数据压缩。requests模块不是Python自带的，所以必须先安装。通过命令行，运行pip install requests（附录A详细介绍了如何安装第三方模块）。

编写requests模块是因为Python的urllib2模块用起来太复杂。实际上，请拿一支记号笔涂黑这一段。忘记我曾提到urllib2。如果你需要从Web下载东西，使用requests模块就好了。

接下来，做一个简单的测试，确保requests模块已经正确安装。在交互式环境中输入以下代码：

```
>>> import requests
```

如果没有错误信息显示，requests模块就已经安装成功了。

11.2.1 用**requests.get()**函数下载一个网页

requests.get()函数接受一个要下载的URL字符串。通过在requests.get()的返回值上调用type()，你可以看到它返回一个Response对象，其中包含了Web服务器对你的请求做出的响应。稍后我将更详细地解释Response对象，但现在请在交互式环境中输入以下代码，并保持计算机与因特网的连接：

```
  >>> import requests
❶ >>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.t
```

```
>>> type(res)
<class 'requests.models.Response'>
❷ >>> res.status_code == requests.codes.ok
True
>>> len(res.text)
178981
>>> print(res.text[:250])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever. You may copy it, give it away or
re-use it under the terms of the Proje
```

该URL指向一个文本页面，其中包含整部罗密欧与朱丽叶，它是由古登堡计划❶提供的。通过检查Response对象的status_code属性，你可以了解对这个网页的请求是否成功。如果该值等于requests.codes.ok，那么一切都好❷（顺便说一下，HTTP协议中“OK”的状态码是200。你可

能已经熟悉404状态码，它表示“没找到”）。

如果请求成功，下载的页面就作为一个字符串，保存在Response对象的text变量中。这个变量保存了包含整部戏剧的一个大字符串，调用len(res.text)表明，它的长度超过178000个字符。最后，调用print(res.text[:250])显示前250个字符。

11.2.2 检查错误

正如你看到的，Response对象有一个status_code属性，可以检查它是否等于requests.codes.ok，了解下载是否成功。检查成功有一种简单的方法，就是在Response对象上调用raise_for_status()方法。如果下载文件出错，这将抛出异常。如果下载成功，就什么也不做。在交互式环境中输入以下代码：

```
>>> res = requests.get('http://inventwithpython.com/page_that_does_not_exis

>>> res.raise_for_status()

Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    res.raise_for_status()
  File "C:\Python34\lib\site-packages\requests\models.py", line 773, in rai
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found
```

raise_for_status()方法是一种很好的方式，确保程序在下载失败时停止。这是一件好事：你希望程序在发生未预期的错误时，马上停止。如果下载失败对程序来说不够严重，可以用try和except语句将

raise_for_status()代码行包裹起来，处理这一错误，不让程序崩溃。

```
import requests
res = requests.get('http://inventwithpython.com/page_that_does_not_exist')
try:
    res.raise_for_status()
except Exception as exc:
    print('There was a problem: %s' % (exc))
```

这次raise_for_status()方法调用导致程序输出以下内容：

```
There was a problem: 404 Client Error: Not Found
```

总是在调用requests.get()之后再调用raise_for_status()。你希望确保下载确实成功，然后再让程序继续。

11.3 将下载的文件保存到硬盘

现在，可以用标准的open()函数和write()方法，将Web页面保存到硬盘中的一个文件。但是，这里稍稍有一点不同。首先，必须用“写二进制”模式打开该文件，即向函数传入字符串'wb'，作为open()的第二参数。即使该页面是纯文本的（例如前面下载的罗密欧与朱丽叶的文本），你也需要写入二进制数据，而不是文本数据，目的是为了保存该文本中的“Unicode编码”。

Unicode编码

Unicode编码超出了本书的范围，但你可以通过以下网页了解更多的相关内容：

- Joel on Software: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!): *http://www.joelonsoftware.com/articles/Unicode.html*
- Pragmatic Unicode: *http://nedbatchelder.com/text/unipain.html*

为了将Web页面写入到一个文件，可以使用for循环和Response对象的iter_content()方法。

```
>>> import requests



>>> res = requests.get('http://www.gutenberg.org/cache/epub/1112/pg1112.txt



>>> res.raise_for_status()



>>> playFile = open('RomeoAndJuliet.txt', 'wb')



>>> for chunk in res.iter_content(100000):



        playFile.write(chunk)



100000
78981
>>> playFile.close()
```

iter_content()方法在循环的每次迭代中，返回一段内容。每一段都是bytes数据类型，你需要指定一段包含多少字节。10万字节通常是不错的选择，所以将100000作为参数传递给iter_content()。

文件RomeoAndJuliet.txt将存在于当前工作目录。请注意，虽然在网站上文件名是pg1112.txt，但在你的硬盘上，该文件的名字不同。requests模块只处理下载网页内容。一旦网页下载后，它就只是程序中的数据。即使在下载该网页后断开了因特网连接，该页面的所有数据仍然会在你的计算机中。

write()方法返回一个数字，表示写入文件的字节数。在前面的例子中，第一段包含100000个字节，文件剩下的部分只需要78981个字节。

回顾一下，下载并保存到文件的完整过程如下：

1．调用requests.get()下载该文件。

2．用'wb'调用open()，以写二进制的方式打开一个新文件。

3．利用Respose对象的iter_content()方法做循环。

4．在每次迭代中调用write()，将内容写入该文件。

5．调用close()关闭该文件。

这就是关于requests模块的全部内容！相对于写入文本文件的open()/write()/ close()工作步骤，for循环和iter_content()的部分可能看起来比较复杂，但这是为了确保requests模块即使在下载巨大的文件时也不会消耗太多内存。你可以访问http://requests.readthedocs.org/，了解requests模块的其他功能。

11.4 HTML

在你拆解网页之前，需要学习一些HTML的基本知识。你也会看到如何利用Web浏览器的强大开发者工具，它们使得从Web抓取信息更容易。

11.4.1 学习HTML的资源

超文本标记语言（HTML）是编写Web页面的格式。本章假定你对HTML有一些基本经验，但如果你需要初学者指南，我推荐以下站点：

- http://htmldog.com/guides/html/beginner/
- http://www.codecademy.com/tracks/web/
- https://developer.mozilla.org/en-US/learn/html/

11.4.2 快速复习

假定你有一段时间没有看过HTML了，这里是对基本知识的快速复习。HTML文件是一个纯文本文件，带有.html文件扩展名。这种文件中的文本被“标签”环绕，标签是尖括号包围的单词。标签告诉浏览器以怎样的格式显示该页面。一个开始标签和一个结束标签可以包围某段文本，形成一个“元素”。“文本”（或“内部的HTML”）是在开始标签和结束标签之间的内容。例如，下面的HTML在浏览器中显示Hello world!，其中Hello用粗体显示。

```
<strong>Hello</strong> world!
```

这段HTML在浏览器中看起来如图11-1所示。



图11-1 浏览器渲染的Hello world!

开始标签< strong>表明，标签包围的文本将使用粗体。结束标签告诉浏览器，粗体文本到此结束。

HTML中有许多不同的标签。有一些标签具有额外的特性，在尖括号内以“属性”的方式展现。例如，< a>标签包含一段文本，它应该是一个链接。这段文本链接的URL是由href属性确定的。下面是一个例子：

```
Al's free &lt;a href="http://inventwithpython.com">Python books&lt;/a>.
```

这段HTML在浏览器中看起来如图11-2所示。

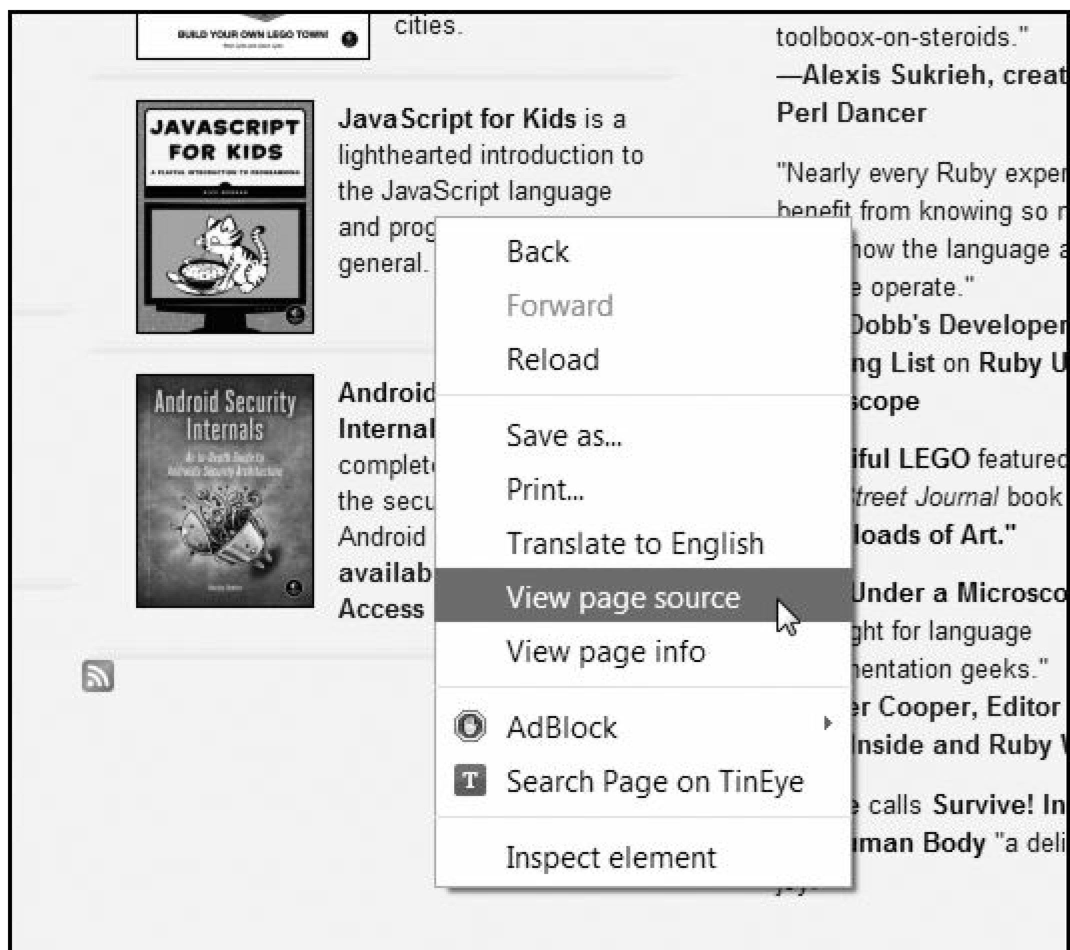


图11-2 浏览器中渲染的链接

某些元素具有id属性，可以用来在页面上唯一地确定该元素。你常常会告诉程序，根据元素的id属性来寻找它。所以利用浏览器的开发者工具，弄清楚元素的id属性，这是编写Web抓取程序常见的任务。

11.4.3 查看网页的**HTML**源代码

对于程序要处理的网页，你需要查看它的HTML源代码。要做到这一点，在浏览器的任意网页上点击右键（或在OS X上Ctrl-点击），选择View Source或View page source，查看该页的 HTML 文本（参见图 11-3）。这是浏览器实际接收到的文本。浏览器知道如何通过这个HTML显示或渲染网页。



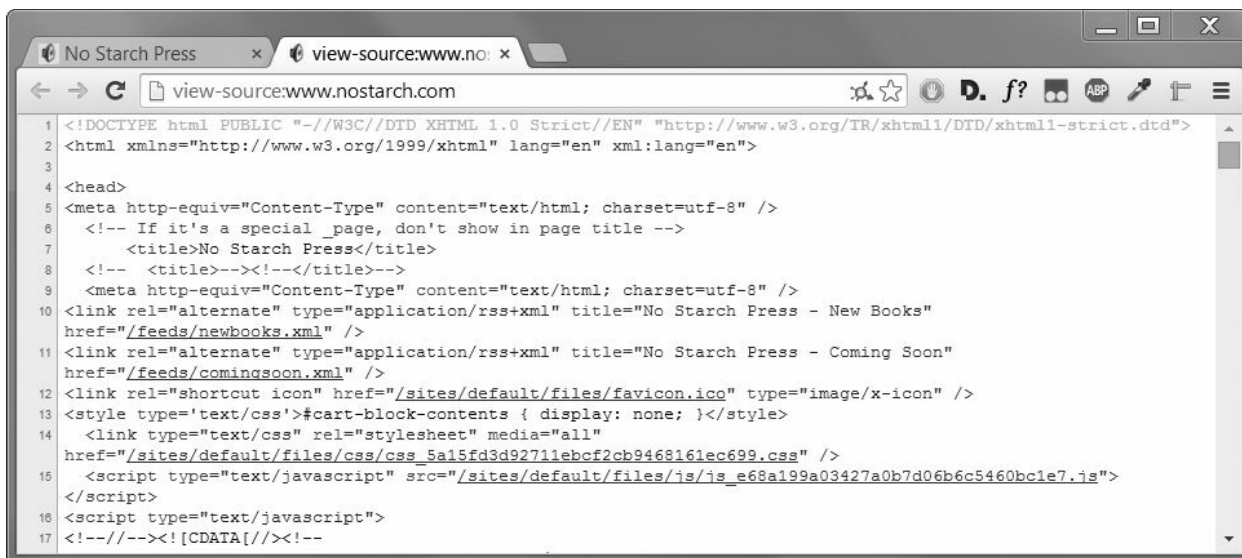


图11-3 查看网页的源代码

我强烈建议你查看一些自己喜欢的网站的HTML源代码。在查看源代码时，如果你不能完全理解，也没有关系。你不需要完全掌握HTML，也能编写简单的Web抓取程序，毕竟你不是要编写自己的网站。只需要足够的知识，就能从已有的网站中挑选数据。

11.4.4 打开浏览器的开发者工具

除了查看网页的源代码，你还可以利用浏览器的开发者工具，来检查页面的HTML。在Windows版的Chrome和IE中，开发者工具已经安装了。可以按下F12，让它们出现（参见图11-4）。再次按下F12，可以让开发者工具消失。在Chrome中，也可以选择View ►Developer ►Developer Tools，调出开发者工具。在OS X中按下⌘- Option-I，将打开Chrome的开发者工具。

对于Firefox，可以在Windows和Linux中需要按下Ctrl-Shift-C，或在OS X中按下⌘-option-C，调出开发者工具查看器。它的布局几乎与Chrome的开发者工具一样。

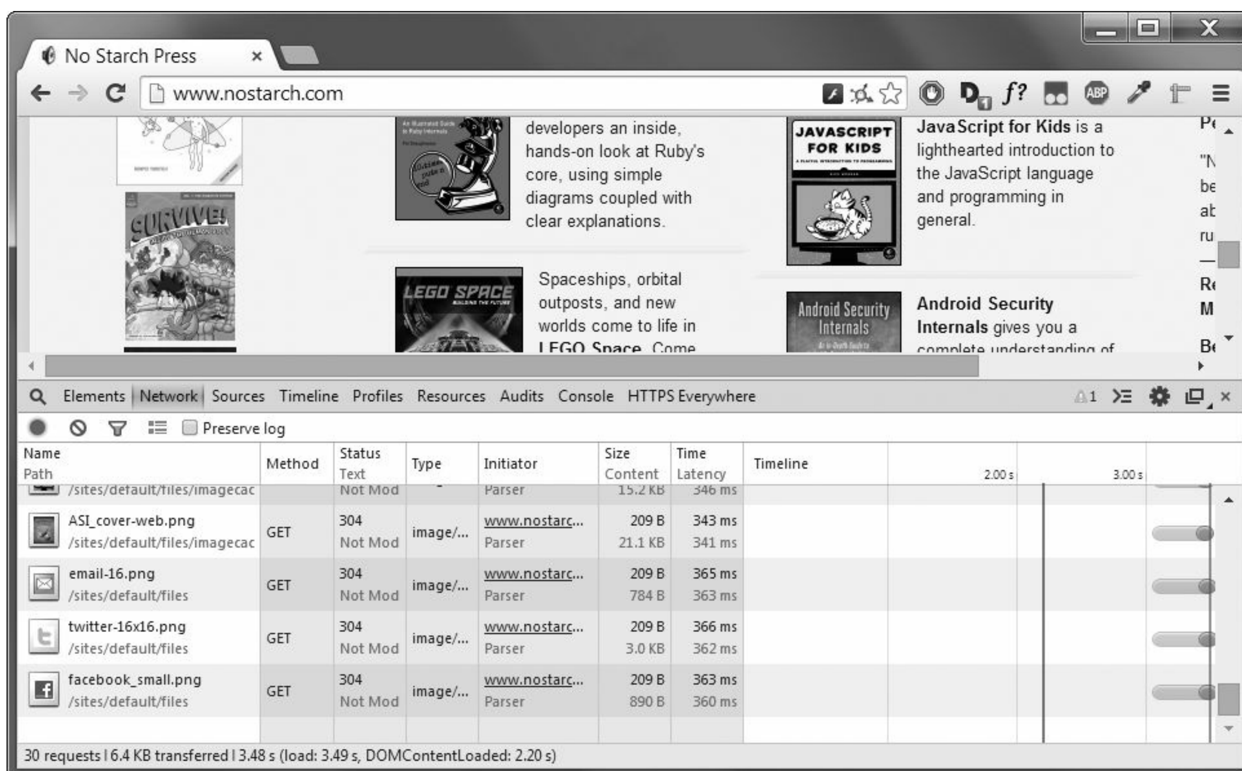


图11-4 Chrome浏览器中的开发者工具窗口

在Safari中，打开Preferences窗口，并在Advanced pane选中Show Develop menu in the menu bar选项。在它启用后，你可以按下⌘-option-I，调出开发者工具。

在浏览器中启用或安装了开发者工具之后，可以在网页中任何部分点击右键，在弹出菜单中选择Inspect Element，查看页面中这一部分对应的HTML。如果需要在Web抓取程序中解析HTML，这很有帮助。

不要用正则表达式来解析HTML

在一个字符串中定位特定的一段HTML，这似乎很适合使用正则表达式。但是，我建议你不要这么做。HTML 的格式可以有许多不同的方式，并且仍然被认为是有效的HTML，但尝试用正则表达式来捕捉所有这些可能的变化，将非常繁琐，并且容易出错。专门用于解析HTML的模块，诸如Beautiful Soup，将更不容易导致缺陷。在*http://stackoverflow.com/a/1732454/1893164/*，你会看到更充分的讨论，了解为什么不应该用正则表达式来解析HTML。

11.4.5 使用开发者工具来寻找HTML元素

程序利用requests模块下载了一个网页之后，你会得到该页的HTML内容，作为一个字符串值。现在你需要弄清楚，这段HTML的哪个部分

对应于网页上你感兴趣的信息。

这就是可以利用浏览器的开发者工具的地方。假定你需要编写一个程序，从http://weather.gov/ 获取天气预报数据。在写代码之前，先做一点调查。如果你访问该网站，并查找邮政编码94105，该网站将打开一个页面，显示该地区的天气预报。

如果你想抓取那个邮政编码对应的气温信息，怎么办？右键点击它在页面的位置（或在OS X上用Control-点击），在弹出的菜单中选择Inspect Element。这将打开开发者工具窗口，其中显示产生这部分网页的HTML。图11-5展示了开发者工具打开显示气温的HTML。

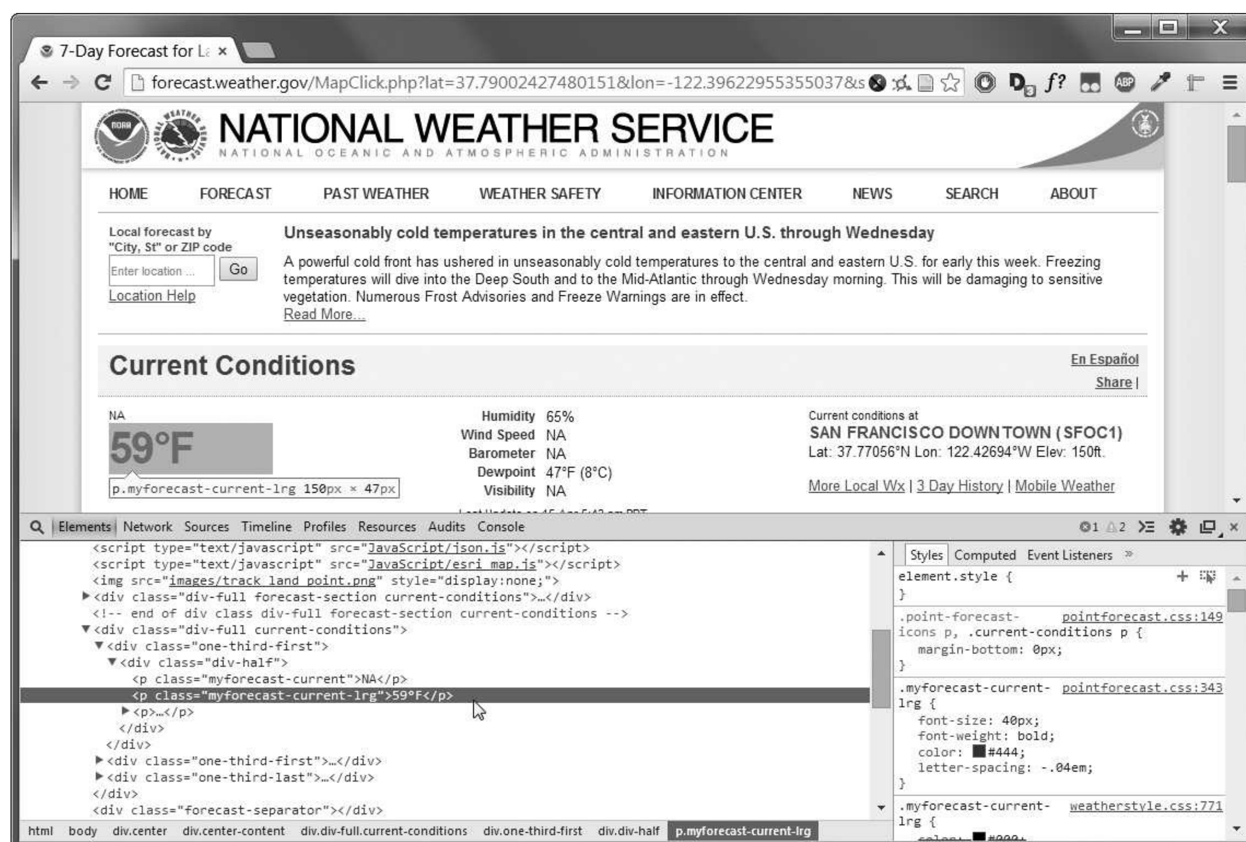


图11-5 用开发者工具查看包含温度文本的元素

通过开发者工具，可以看到网页中负责气温部分的 HTML 是< p class= "myforecast- current-lrg">57°F< /p>。这正是你要找的东西！看起来气温信息包含在一个< p>元素中，带有myforecast-current-lrg类。既然你知道了要找的是什么，BeautifulSoup模块就可以帮助你在这个字符串中找到它。

11.5 用BeautifulSoup模块解析HTML

Beautiful Soup是一个模块，用于从HTML页面中提取信息（用于这个目的时，它比正则表达式好很多）。BeautifulSoup模块的名称是bs4（表示Beautiful Soup，第4版）。要安装它，需要在命令行中运行pip install beautifulsoup4（关于安装第三方模块的指导，请查看附录 A）。虽然安装时使用的名字是beautifulsoup4，但要导入它，就使用import bs4。

在本章中，Beautiful Soup的例子将解析（即分析并确定其中的一些部分）硬盘上的一个HTML文件。在IDLE中打开一个新的文件编辑器窗口，输入以下代码，并保存为example.html。或者，从http://nostarch.com/automatestuff/下载它。

```
<!-- This is the example.html example file. -->

<html><head><title>The Website Title</title></head>
<body>
<p>Download my <strong>Python</strong> book from <a href="http://
inventwithpython.com">my website</a>.</p>
<p class="slogan">Learn Python the easy way!</p>
<p>By <span id="author">Al Sweigart</span></p>
</body></html>
```

你可以看到，既使一个简单的HTML文件，也包含许多不同的标签和属性。对于复杂的网站，事情很快就变得令人困惑。好在，Beautiful Soup让处理HTML变得容易很多。

11.5.1 从HTML创建一个BeautifulSoup对象

bs4.BeautifulSoup()函数调用时需要一个字符串，其中包含将要解析的HTML。bs4.BeautifulSoup()函数返回一个BeautifulSoup对象。在交互式环境中输入以下代码，同时保持计算机与因特网的连接：

```
>>> import requests, bs4
```

```
>>> res = requests.get('http://nostarch.com')
>>> res.raise_for_status()
>>> noStarchSoup = bs4.BeautifulSoup(res.text)
>>> type(noStarchSoup)
< class 'bs4.BeautifulSoup'>
```

这段代码利用requests.get()函数从No Starch Press网站下载主页，然后将响应结果的text属性传递给bs4.BeautifulSoup()。它返回的BeautifulSoup对象保存在变量noStarchSoup中。

也可以向bs4.BeautifulSoup()传递一个File对象，从硬盘加载一个HTML文件。在交互式环境中输入以下代码（确保example.html文件在工作目录中）：

```
>>> exampleFile = open('example.html')
```

```
>>> exampleSoup = bs4.BeautifulSoup(exampleFile)

>>> type(exampleSoup)

< class 'bs4.BeautifulSoup'>
```

有了BeautifulSoup对象之后，就可以利用它的方法，定位HTML文档中的特定部分。

11.5.2 用select()方法寻找元素

针对你要寻找的元素，调用method()方法，传入一个字符串作为CSS“选择器”，这样就可以取得Web页面元素。选择器就像正则表达式：它们指定了要寻找的模式，在这个例子中，是在HTML页面中寻找，而不是普通的文本字符串。

完整地讨论 CSS 选择器的语法超出了本书的范围（在http://nostarch.com/automatestuff/的资源中，有很好的选择器指南），但这里有一份选择器的简单介绍。表11-2举例展示了大多数常用CSS选择器的模式。

表**11-2 CSS**选择器的例子

传递给**select()** 方法的选择器	将匹配…

soup.select('div')	所有名为< div>的元素
soup.select('#author')	带有id属性为author的元素
soup.select('.notice')	所有使用CSS class属性名为notice的元素
soup.select('div span')	所有在< div>元素之内的< span>元素
soup.select('div > span')	所有直接在< div>元素之内的< span>元素，中间没有其他元素
soup.select('input[name]')	所有名为< input>，并有一个name属性，其值无所谓的元素
soup.select('input[type="button"]')	所有名为< input>，并有一个type属性，其值为button的元素

不同的选择器模式可以组合起来，形成复杂的匹配。例如，soup.select('p #author')将匹配所有id属性为author的元素，只要它也在一个< p>元素之内。

select()方法将返回一个Tag对象的列表，这是Beautiful Soup表示一个HTML元素的方式。针对BeautifulSoup对象中的HTML的每次匹配，列表中都有一个Tag对象。Tag值可以传递给str()函数，显示它们代表的HTML标签。Tag值也可以有attrs属性，它将该Tag的所有HTML属性作为一个字典。利用前面的example.html文件，在交互式环境中输入以下代码：

```
>>> import bs4



>>> exampleFile = open('example.html')
```

```
>>> exampleSoup = bs4.BeautifulSoup(exampleFile.read())

>>> elems = exampleSoup.select('#author')

>>> type(elems)

< class 'list'>
>>> len(elems)

1
>>> type(elems[0])

< class 'bs4.element.Tag'>
>>> elems[0].getText()

'Al Sweigart'
>>> str(elems[0])
```

```
'< span id="author">Al Sweigart< /span>'
>>> elems[0].attrs
{'id': 'author'}
```

这段代码将带有 id="author"的元素，从示例 HTML 中找出来。我们使用select('#author')返回一个列表，其中包含所有带有id="author"的元素。我们将这个Tag对象的列表保存在变量中elems，len(elems)告诉我们列表中只有一个Tag对象，只有一次匹配。在该元素上调用getText()方法，返回该元素的文本，或内部的HTML。一个元素的文本是在开始和结束标签之间的内容：在这个例子中，就是'Al Sweigart'。

将该元素传递给str()，这将返回一个字符串，其中包含开始和结束标签，以及该元素的文本。最后，attrs给了我们一个字典，包含该元素的属性'id'，以及id属性的值'author'。

也可以从BeautifulSoup对象中找出< p>元素。在交互式环境中输入以下代码：

```
>>> pElems = exampleSoup.select('p')
>>> str(pElems[0])
```

```
'< p>Download my < strong>Python< /strong> book from < a href="http://
inventwithpython.com">my website< /a>.< /p>'
>>> pElems[0].getText()
'Download my Python book from my website.'
>>> str(pElems[1])
'< p class="slogan">Learn Python the easy way!< /p>'
>>> pElems[1].getText()
'Learn Python the easy way!'
>>> str(pElems[2])
'< p>By < span id="author">Al Sweigart< /span>< /p>'
>>> pElems[2].getText()
'By Al Sweigart'
```

这一次，select()给我们一个列表，包含3次匹配，我们将它保存在pElems中。在pElems[0]、pElems[1]和pElems[2]上使用str()，将每个元素显示为一个字符串，并在每个元素上使用getText()，显示它的文本。

11.5.3 通过元素的属性获取数据

Tag对象的get()方法让我们很容易从元素中获取属性值。向该方法传入一个属性名称的字符串，它将返回该属性的值。利用example.html，在交互式环境中输入以下代码：

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example.html'))
>>> spanElem = soup.select('span')[0]
>>> str(spanElem)
'< span id="author">Al Sweigart< /span>'
>>> spanElem.get('id')
'author'
>>> spanElem.get('some_nonexistent_addr') == None
True
>>> spanElem.attrs
```

```
{'id': 'author'}
```

这里，我们使用select()来寻找所有< span>元素，然后将第一个匹配的元素保存在spanElem中。将属性名'id'传递给get()，返回该属性的值'author'。

11.6 项目：“I’m Feeling Lucky”Google查找

每次我在 Google 上搜索一个主题时，都不会一次只看一个搜索结果。通过鼠标中键点击搜索结果链接，或在点击时按住CTRL键，我会在一些新的选项卡中打开前几个链接，稍后再来查看。我经常搜索Google，所以这个工作流程（开浏览器，查找一个主题，依次用中键点击几个链接）变得很乏味。如果我只要在命令行中输入查找主题，就能让计算机自动打开浏览器，并在新的选项卡中显示前面几项查询结果，那就太好了。让我们写一个脚本来完成这件事。

下面是程序要做的事：

- 从命令行参数中获取查询关键字。
- 取得查询结果页面。
- 为每个结果打开一个浏览器选项卡。

这意味着代码需要完成以下工作：

- 从sys.argv中读取命令行参数。
- 用requests模块取得查询结果页面。
- 找到每个查询结果的链接。
- 调用webbrowser.open()函数打开Web浏览器。

打开一个新的文件编辑器窗口，并保存为lucky.py。

第 1 步：获取命令行参数，并请求查找页面

开始编码之前，你首先要知道查找结果页面的URL。在进行Google查找后，你看浏览器地址栏，就会发现结果页面的URL类似于https://www.google.com/ search?q=SEARCH_TERM_HERE。requests模块可以下载这个页面，然后可以用Beautiful Soup，找到HTML中的查询结果的链接。最后，用webbrowser模块，在浏览器选项卡中打开这些链接。

让你的代码看起来像这样：

```
#! python3
# lucky.py - Opens several Google search results.

import requests, sys, webbrowser, bs4

print('Googling...')     # display text while downloading the Google page
res = requests.get('http://google.com/search?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

用户运行该程序时，将通过命令行参数指定查找的主题。这些参数将作为字符串，保存在sys.argv列表中。

第 2 步：找到所有的结果

现在你需要使用Beautiful Soup，从下载的HTML中，提取排名靠前的查找结果链接。但如何知道完成这项工作需要怎样的选择器？例如，你不能只查找所有的< a>标签，因为在这个HTML中，有许多链接你是不关心的。因此，必须用浏览器的开发者工具来检查这个查找结果页面，尝试寻找一个选择器，它将挑选出你想要的链接。

在针对Beautiful Soup进行Google查询后，你可以打开浏览器的开发者工具，查看该页面上的一些链接元素。它们看起来复杂得难以置信，大概像这样：< a href="/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCgQFjAA&url=http%3A%2F%2Fw%2FBeautifulSoup%2F&ei=LHBVU_XDD9KVyAShmYDwCw&usg=AFQjCNHAxwplurFTuLQ&sig2=sdZu6WVlBlVSDrwhtworMA" onmousedown="return rwt(this,'','','','1','AFQjCNH AxwplurFOBqg5cehWQEVKi-TuLQ','sdZu6WVlBlVSDrwhtworMA','0CCgQFjAA','','',event)" data-href="http://www. crummy.com/software/BeautifulSoup/">< em> BeautifulSoup< /em>: We called him Tortoise because he taught us.< /a>

该元素看起来复杂得难以置信，但这没有关系。只需要找到查询结果链接都具有的模式。但这个< a>元素没有什么特殊，难以和该页面上非查询结果的< a>元素区分开来。

确保你的代码看起来像这样：

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip

--

# Retrieve top search result links.

soup = bs4.BeautifulSoup(res.text)
```

```
# Open a browser tab for each result.



linkElems = soup.select('.r a')
```

但是，如果从< a>元素向上看一点，就会发现这样一个元素：< h3 class="r">。查看余下的HTML源代码，看起来r类仅用于查询结果链接。你不需要知道CSS类r是什么，或者它会做什么。只需要利用它作为一个标记，查找需要的< a>元素。可以通过下载页面的HTML文本，创建一个BeautifulSoup对象，然后用选择符'.r a'，找到所有具有CSS类r的元素中的< a>元素。

第**3**步：针对每个结果打开**Web**浏览器

最后，我们将告诉程序，针对结果打开Web浏览器选项卡。将下面的内容添加到程序的末尾：

```
#! python3
# lucky.py - Opens several google search results.

import requests, sys, webbrowser, bs4

--snip



--
```

```
# Open a browser tab for each result.
linkElems = soup.select('.r a')
numOpen = min(5, len(linkElems))
for i in range(numOpen):
    webbrowser.open('http://google.com' + linkElems[i].get('href'))
```

默认情况下，你会使用webbrowser模块，在新的选项卡中打开前5个查询结果。但是，用户查询的主题可能少于5个查询结果。soup.select()调用返回一个列表，包含匹配'.r a'选择器的所有元素，所以打开选项卡的数目要么是5，要么是这个列表的长度（取决于哪一个更小）。

内建的Python函数min()返回传入的整型或浮点型参数中最小的一个（也有内建的max()函数，返回传入的参数中最大的一个）。你可以使用min()弄清楚该列表中是否少于5个链接，并且将要打开的链接数保存在变量numOpen中。然后可以调用range(numOpen)，执行一个for循环。

在该循环的每次迭代中，你使用webbrowser.open()，在Web浏览器中打开一个新的选项卡。请注意，返回的< a>元素的href属性中，不包含初始的http://google.com部分，所以必须连接它和href属性的字符串。

现在可以马上打开前5个Google查找结果，比如说，要查找Python programming tutorials，你只要在命令行中运行lucky python programming

tutorials（如何在你的操作系统中方便地运行程序，请参看附录B）。

第4步：类似程序的想法

分选项卡浏览的好处在于，很容易在新选项卡中打开一些链接，稍后再来查看。一个自动打开几个链接的程序，很适合快捷地完成下列任务：

- 查找亚马逊这样的电商网站后，打开所有的产品页面；
- 打开针对一个产品的所有评论的链接；
- 查找Flickr或Imgur这样的照片网站后，打开查找结果中的所有照片的链接。

11.7 项目：下载所有XKCD漫画

博客和其他经常更新的网站通常有一个首页，其中有最新的帖子，以及一个“前一篇”按钮，将你带到以前的帖子。然后那个帖子也有一个“前一篇”按钮，以此类推。这创建了一条线索，从最近的页面，直到该网站的第一个帖子。如果你希望拷贝该网站的内容，在离线的时候阅读，可以手工导航至每个页面并保存。但这是很无聊的工作，所以让我们写一个程序来做这件事。

XKCD 是一个流行的极客漫画网站，它符合这个结构（参见图 11-6）。首页http://xkcd.com/ 有一个“Prev”按钮，让用户导航到前面的漫画。手工下载每张漫画要花较长的时间，但你可以写一个脚本，在几分钟内完成这件事。

下面是程序要做的事：

- 加载主页；
- 保存该页的漫画图片；
- 转入前一张漫画的链接；
- 重复直到第一张漫画。

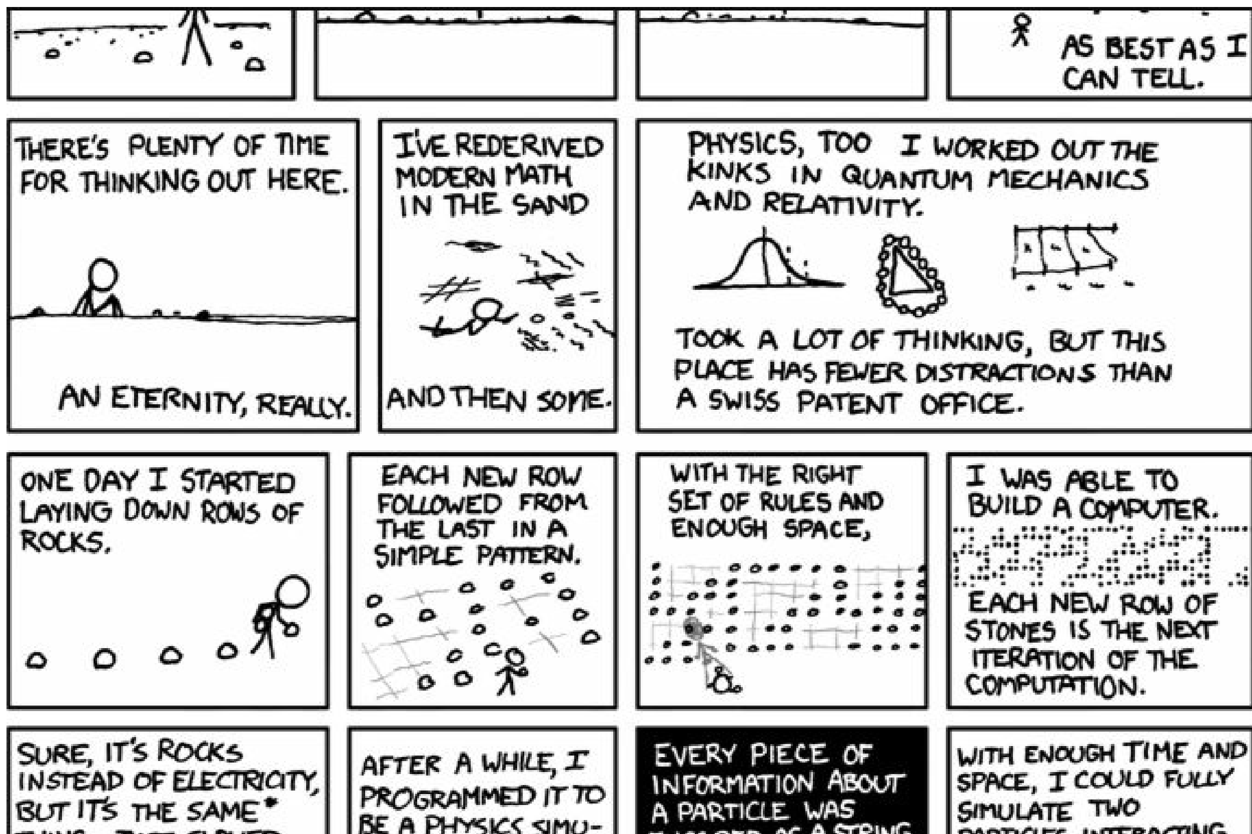


图11-6 XKCD，“关于浪漫、讽刺、数学和语言的漫画网站”

这意味着代码需要做下列事情：

- 利用requests模块下载页面。
- 利用Beautiful Soup找到页面中漫画图像的URL。
- 利用iter_content()下载漫画图像，并保存到硬盘。
- 找到前一张漫画的链接URL，然后重复。

打开一个新的文件编辑器窗口，将它保存为downloadXkcd.py。

第1步：设计程序

打开一个浏览器的开发者工具，检查该页面上的元素，你会发现下面的内容：

- 漫画图像文件的URL，由一个< img>元素的href属性给出。
- < img>元素在< div id="comic">元素之内。
- Prev按钮有一个rel HTML属性，值是prev。

- 第一张漫画的Prev按钮链接到http://xkcd.com/# URL，表明没有前一个页面了。

让你的代码看起来像这样：

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'               # starting url
os.makedirs('xkcd', exist_ok=True)    # store comics in ./xkcd
while not url.endswith('#'):
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

你会有一个url变量，开始的值是'http://xkcd.com'，然后反复更新（在一个for循环中），变成当前页面的Prev链接的URL。在循环的每一步，你将下载URL上的漫画。如果URL以'#'结束，你就知道需要结束循环。

将图像文件下载到当前目录的一个名为xkcd的文件夹中。调用os.makedirs()函数。确保这个文件夹存在，并且关键字参数exist_ok=True在该文件夹已经存在时，防止该函数抛出异常。剩下的代码只是注释，列出了剩下程序的大纲。

第2步：下载网页

我们来实现下载网页的代码。让你的代码看起来像这样：

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

url = 'http://xkcd.com'                # starting url
os.makedirs('xkcd', exist_ok=True)     # store comics in ./xkcd
while not url.endswith('#'):
    # Download the page.




    print('Downloading page %s...' % url)




    res = requests.get(url)




    res.raise_for_status()




    soup = bs4.BeautifulSoup(res.text)




    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.
```

```
print('Done.')
```

首先，打印url，这样用户就知道程序将要下载哪个URL。然后利用requests模块的request.get()函数下载它。像以往一样，马上调用Response对象的raise_for_status()方法，如果下载发生问题，就抛出异常，并终止程序。否则，利用下载页面的文本创建一个BeautifulSoup对象。

第3步：寻找和下载漫画图像

让你的代码看起来像这样：

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip

--

    # Find the URL of the comic image.

    comicElem = soup.select('#comic img')

    if comicElem == []:
```

```
        print('Could not find comic image.')

    else:

        comicUrl = 'http:' comicElem[0].get('src')

        # Download the image.

        print('Downloading image %s...' % (comicUrl))

        res = requests.get(comicUrl)

        res.raise_for_status()

    # TODO: Save the image to ./xkcd.
```

```
    # TODO: Get the Prev button's url.

print('Done.')
```

用开发者工具检查XKCD主页后，你知道漫画图像的< img>元素是在一个< div>元素中，它带有的id属性设置为comic。所以选择器'#comic img'将从BeautifulSoup对象中选出正确的< img>元素。

有一些XKCD页面有特殊的内容，不是一个简单的图像文件。这没问题，跳过它们就好了。如果选择器没有找到任何元素，那么soup.select('#comic img')将返回一个空的列表。出现这种情况时，程序将打印一条错误消息，不下载图像，继续执行。

否则，选择器将返回一个列表，包含一个< img>元素。可以从这个< img>元素中取得src属性，将它传递给requests.get()，下载这个漫画的图像文件。

第4步：保存图像，找到前一张漫画

让你的代码看起来像这样：

```
#! python3
# downloadXkcd.py - Downloads every single XKCD comic.

import requests, os, bs4

--snip

--

        # Save the image to ./xkcd.
```

```
        imageFile = open(os.path.join('xkcd', os.path.basename(comicUrl)),

        for chunk in res.iter_content(100000):

            imageFile.write(chunk)

        imageFile.close()


    # Get the Prev button's url.

    prevLink = soup.select('a[rel="prev"]')[0]

    url = 'http://xkcd.com' + prevLink.get('href')


print('Done.')
```

这时，漫画的图像文件保存在变量res中。你需要将图像数据写入硬盘的文件。

你需要为本地图像文件准备一个文件名，传递给 open()。comicUrl的值类似'http://imgs.xkcd.com/comics/heartbleed_explanation.png'。你可能注意到，它看起来很像文件路径。实际上，调用os.path.basename()时传入comicUrl，它只返回URL的最后部分：'heartbleed_explanation.png'。你可以用它作为文件名，将图像保存到硬盘。用os.path.join()连接这个名称和xkcd文件夹的名称，这样程序就会在Windows下使用倒斜杠（\），在OS X和Linux下使用斜杠（/）。既然你最后得到了文件名，就可以调用open()，用'wb'模式打开一个新文件。

回忆一下本章早些时候，保存利用 Requests 下载的文件时，你需要循环处理iter_content()方法的返回值。for循环中的代码将一段图像数据写入文件（每次最多10万字节），然后关闭该文件。图像现在保存到硬盘中。

然后，选择器'a[rel="prev"]'识别出rel属性设置为prev的< a>元素，利用这个< a>元素的href属性，取得前一张漫画的URL，将它保存在url中。然后while循环针对这张漫画，再次开始整个下载过程。

这个程序的输出看起来像这样：

```
Downloading page http://xkcd.com...
Downloading image http://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page http://xkcd.com/1358/...
Downloading image http://imgs.xkcd.com/comics/nro.png...
Downloading page http://xkcd.com/1357/...
Downloading image http://imgs.xkcd.com/comics/free_speech.png...
Downloading page http://xkcd.com/1356/...
Downloading image http://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page http://xkcd.com/1355/...
Downloading image http://imgs.xkcd.com/comics/airplane_message.png...
Downloading page http://xkcd.com/1354/...
Downloading image http://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip
```

--

这个项目是一个很好的例子，说明程序可以自动顺着链接，从网络上抓取大量的数据。你可以从Beautiful Soup的文档了解它的更多功能：http://www. crummy.com/ software/BeautifulSoup/bs4/doc/.

第5步：类似程序的想法

下载页面并追踪链接，是许多网络爬虫程序的基础。类似的程序也可以做下面的事情：

- 顺着网站的所有链接，备份整个网站。
- 拷贝一个论坛的所有信息。
- 复制一个在线商店中所有产品的目录。

requests 和 BeautifulSoup 模块很了不起，只要你能弄清楚需要传递给requests.get()的URL。但是，有时候这并不容易找到。或者，你希望编程浏览的网站可能要求你先登录。selenium 模块将让你的程序具有执行这种复杂任务的能力。

11.8 用selenium模块控制浏览器

selenium模块让Python直接控制浏览器，实际点击链接，填写登录信息，几乎就像是有一个人类用户在与页面交互。与Requests和Beautiful Soup相比，Selenium允许你用高级得多的方式与网页交互。但因为它启动了Web浏览器，假如你只是想从网络上下载一些文件，会有点慢，并且难以在后台运行。

附录A有安装第三方模块的详细步骤。

11.8.1 启动selenium控制的浏览器

对于这些例子，你需要FireFox浏览器。它将成为你控制的浏览器。如果你还没有FireFox，可以从http://getfirefox.com/免费下载它。

导入selenium的模块需要一点技巧。不是import selenium，而是要运行from selenium import webdriver（为什么selenium模块要使用这种方式设置？答案超出了本书的范围）。之后，你可以用selenium启动FireFox浏览器。在交互式环境中输入以下代码：

```
>>> from selenium import webdriver




>>> browser = webdriver.Firefox()




>>> type(browser)




< class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('http://inventwithpython.com')
```

你会注意到，当 webdriver.Firefox()被调用时，FireFox 浏览器启动了。对值webdriver.Firefox()调用 type()，揭示它具有 WebDriver 数据类

型。调用 browser. get('http://inventwithpython.com')将浏览器指向http://inventwithpython.com/。浏览器应该看起来如图11-7所示。



图11-7 在IDLE中调用webdriver.Firefox()和get()后，FireFox浏览器出现了

11.8.2 在页面中寻找元素

WebDriver对象有好几种方法，用于在页面中寻找元素。它们被分成find*element* 和*find* elements方法。find*element* 方法返回一个*WebElement*对象，代表页面中匹配查询的第一个元素。*find* elements方法返回WebElement_*对象的列表，包含页面中所有匹配的元素。

表11-3展示了find*element* 和*find* elements方法的几个例子，它们在变量browser中保存的WebDriver对象上调用。

表**11-3 selenium**的**WebDriver**方法，用于寻找元素

方法名	返回的**WebElement**对象/列表
browser.find_element_by_class_name(*name*) browser.find_elements_by_class_name(*name*)	使用CSS类name的元素

browser.find_element_by_css_selector(*selector*) browser.find_elements_by_css_selector(*selector*)	匹配CSS *selector* 的元素
browser.find_element_by_id(*id*) browser.find_elements_by_id(*id*)	匹配*id* 属性值的元素
browser.find_element_by_link_text(*text*) browser.find_elements_by_link_text(*text*)	完全匹配提供的text的< a>元素
browser.find_element_by_partial_link_text(*text*) browser.find_elements_by_partial_link_text(*text*)	包含提供的text的< a>元素
browser.find_element_by_name(*name*) browser.find_elements_by_name(*name*)	匹配*name* 属性值的元素
browser.find_element_by_tag_name(*name*) browser.find_elements_by_tag_name(*name*)	匹配标签*name* 的元素 (大小写无关，< a>元素匹配'a'和'A')

除了*_by_tag_name()方法，所有方法的参数都是区分大小写的。如果页面上没有元素匹配该方法要查找的元素，selenium模块就会抛出NoSuchElement异常。如果你不希望这个异常让程序崩溃，就在代码中添加try和except语句。

一旦有了WebElement对象，就可以读取表11-4中的属性，或调用其中的方法，了解它的更多功能。

表**11-4 WebElement**的属性和方法

属性或方法	描述
tag_name	标签名，例如 'a'表示< a>元素
get_attribute(*name*)	该元素name属性的值

text	该元素内的文本，例如< span>hello< /span>中的'hello'
clear()	对于文本字段或文本区域元素，清除其中输入的文本
is_displayed()	如果该元素可见，返回True，否则返回False
is_enabled()	对于输入元素，如果该元素启用，返回True，否则返回False
is_selected()	对于复选框或单选框元素，如果该元素被选中，选择True，否则返回False
location	一个字典，包含键'x'和'y'，表示该元素在页面上的位置

例如，打开一个新的文件编辑器，输入以下程序：

```
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://inventwithpython.com')
try:
    elem = browser.find_element_by_class_name('bookcover')
    print('Found < %s> element with that class name!' % (elem.tag_name))
except:
    print('Was not able to find an element with that name.')
```

这里我们打开FireFox，让它指向一个URL。在这个页面上，我们试图找到带有类名'bookcover'的元素。如果找到这样的元素，我们就用tag_name属性将它的标签名打印出来。如果没有找到这样的元素，就打印不同的信息。

这个程序的输出如下：

```
Found < img> element with that class name!
```

我们发现了一个元素带有类名'bookcover'，它的标签名是'img'。

11.8.3 点击页面

find*element* 和*find* elements方法返回的WebElement对象有一个click()方法，模拟鼠标在该元素上点击。这个方法可以用于链接跳转，选择单选按钮，点击提交按钮，或者触发该元素被鼠标点击时发生的任何事情。例如，在交互式环境中输入以下代码：

```
>>> from selenium import webdriver

>>> browser = webdriver.Firefox()

>>> browser.get('http://inventwithpython.com')

>>> linkElem = browser.find_element_by_link_text('Read It Online')

>>> type(linkElem)
```

```
< class 'selenium.webdriver.remote.webelement.WebElement'>
>>> linkElem.click()     # follows the "Read It Online" link
```

这段程序打开FireFox，指向http://inventwithpython.com/，取得< a>元素的WebElement对象，它的文本是“Read It Online”，然后模拟点击这个元素。就像你自己点击这个链接一样，浏览器将跳转到这个链接。

11.8.4 填写并提交表单

向Web页面的文本字段发送击键，只要找到那个文本字段的< input>或< textarea>元素，然后调用send_keys()方法。例如，在交互式环境中输入以下代码：

```
<code>>>> from selenium import webdriver

>>> browser = webdriver.Firefox()

>>> browser.get('http://gmail.com')

>>> emailElem = browser.find_element_by_id('Email')
```

```
>>> emailElem.send_keys('not_my_real_email@gmail.com')

>>> passwordElem = browser.find_element_by_id('Passwd')

>>> passwordElem.send_keys('12345')

>>> passwordElem.submit()
```

只要Gmail没有在本书出版后改变Username和Password文本字段的id，上面的代码就会用提供的文本填写这些文本字段（你总是可以用浏览器的开发者工具验证id）。在任何元素上调用submit()方法，都等同于点击该元素所在表单的Submit按钮（你可以很容易地调用emailElem.submit()，代码所做的事情一样）。

11.8.5 发送特殊键

selenium有一个模块，针对不能用字符串值输入的键盘击键。它的

功能非常类似于转义字符。这些值保存在selenium.webdriver.common.keys模块的属性中。由于这个模块名非常长，所以在程序顶部运行from selenium.webdriver. common.keys import Keys就比较容易。如果这么做，原来需要写from selenium. webdriver.common.keys的地方，就只要写Keys。表11-5列出了常用的Keys变量。

表**11-5 selenium.webdriver.common.keys**模块中常用的变量

属性	含义
Keys.DOWN, Keys.UP, Keys.LEFT,Keys.RIGHT	键盘箭头键
Keys.ENTER, Keys.RETURN	回车和换行键
Keys.HOME, Keys.END, Keys.PAGE_DOWN,Keys.PAGE_UP	Home键、End键、PageUp键和Page Down键
Keys.ESCAPE, Keys.BACK_SPACE,Keys.DELETE	Esc、Backspace和字母键
Keys.F1, Keys.F2, . . . , Keys.F12	键盘顶部的F1到F12键
Keys.TAB	Tab键

例如，如果光标当前不在文本字段中，按下home和end键，将使浏览器滚动到页面的顶部或底部。在交互式环境中输入以下代码，注意send_keys()调用是如何滚动页面的：

```
>>> from selenium import webdriver
```

```
>>> from selenium.webdriver.common.keys import Keys

>>> browser = webdriver.Firefox()

>>> browser.get('http://nostarch.com')

>>> htmlElem = browser.find_element_by_tag_name('html')

>>> htmlElem.send_keys(Keys.END)     # scrolls to bottom

>>> htmlElem.send_keys(Keys.HOME)     # scrolls to top
```

< html>标签是HTML文件中的基本标签：HTML文件的完整内容包含在< html>和< /html>标签之内。调用browser.find_element_by_tag_name('html')是像一般Web页面发送按键的

好地方。当你滚动到该页的底部，新的内容就会加载，这可能会有用。

11.8.6 点击浏览器按钮

利用以下的方法，selenium也可以模拟点击各种浏览器按钮：

browser.back()点击“返回”按钮。

browser.forward()点击“前进”按钮。

browser.refresh()点击“刷新”按钮。

browser.quit()点击“关闭窗口”按钮。

11.8.7 关于selenium的更多信息

selenium能做的事远远超出了这里描述的功能。它可以修改浏览器的cookie，截取页面快照，运行定制的JavaScript。要了解这些功能的更多信息，请参考文档：http://selenium-python.readthedocs.org/。

11.9 小结

大多数无聊的任务并不限于操作你计算机中的文件。能够编程下载网页，可以让你的程序扩展到因特网。requests模块让下载变得很简单，加上HTML的概念和选择器的基本知识，你就可以利用BeautifulSoup模块，解析下载的网页。

但要全面自动化所有针对网页的任务，你需要利用selenium模块，直接控制Web浏览器。selenium模块将允许你自动登录到网站，填写表单。因为Web浏览器是在因特网上收发信息的最常见方式，所以这是程序员工具箱中一件了不起的工具。

11.10 习题

1．简单描述webbrowser、requests、BeautifulSoup和selenium模块之间的不同。

2．requests.get()返回哪种类型的对象？如何以字符串的方式访问下载的内容？

3．哪个Requests方法检查下载是否成功？

4．如何取得Requests响应的HTTP状态码？

5．如何将Requests响应保存到文件？

6．要打开浏览器的开发者工具，快捷键是什么？

7．在开发者工具中，如何查看页面上特定元素的HTML？

8．要找到id属性为main的元素，CSS选择器的字符串是什么？

9．要找到CSS类为highlight的元素，CSS选择器的字符串是什么？

10．要找到一个< div>元素中所有的< div>元素，CSS 选择器的字符串是什么？

11．要找到一个< button>元素，它的value属性被设置为favorite，CSS选择器的字符串是什么？

12．假定你有一个Beautiful Soup的Tag对象保存在变量spam中，针对的元素是< div>Hello world!< /div>。如何从这个Tag对象中取得字符串'Hello world!'？

13．如何将一个Beautiful Soup的Tag对象的所有属性保存到变量linkElem中？

14．运行import selenium没有效果。如何正确地导入selenium模块？

15．find*element* 和*find* elements方法之间的区别是什么？

16．Selenium的WebElement对象有哪些方法来模拟鼠标点击和键盘击键？

17．你可以在Submit按钮的WebElement对象上调用send_keys(Keys.ENTER)，但利用selenium，还有什么更容易的方法提交

表单？

18．利用selenium如何模拟点击浏览器的“前进”、“返回”和“刷新”按钮？

11.11 实践项目

作为实践，编程完成下列任务。

11.11.1 命令行邮件程序

编写一个程序，通过命令行接受电子邮件地址和文本字符串。然后利用selenium登录到你的邮件账号，将该字符串作为邮件，发送到提供的地址（你也许希望为这个程序建立一个独立的邮件账号）。

这是为程序添加通知功能的一种好方法。你也可以编写类似的程序，从Facebook或Twitter账号发送消息。

11.11.2 图像网站下载

编写一个程序，访问图像共享网站，如Flickr或Imgur，查找一个类型的照片，然后下载所有查询结果的图像。可以编写一个程序，访问任何具有查找功能的图像网站。

11.11.3 2048

2048是一个简单的游戏，通过箭头向上、下、左、右移动滑块，让滑块合并。实际上，你可以通过一遍一遍的重复“上、右、下、左”模式，获得相当高的分数。编写一个程序，打开https://gabrielecirulli.github.io/2048/上的游戏，不断发送上、右、下、左按键，自动玩游戏。

11.11.4 链接验证

编写一个程序，对给定的网页URL，下载该页面所有链接的页面。程序应该标记出所有具有404“Not Found”状态码的页面，将它们作为坏链接输出。

[1] 答案是没有。

第12章　处理Excel电子表格

Excel是Windows环境下流行的、强大的电子表格应用。openpyxl模块让Python程序能读取和修改Excel电子表格文件。例如，可能有一个无聊的任务，需要从一个电子表格拷贝一些数据，粘贴到另一个电子表格中。或者可能需要从几千行中挑选几行，根据某种条件稍作修改。或者需要查看几百份部门预算电子表格，寻找其中的赤字。正是这种无聊无脑的电子表格任务，可以通过Python来完成。

LibreOffice Calc和OpenOffice Calc都能处理Excel的电子表格文件格式，这意味着 openpyxl 模块也能处理来自这些应用程序的电子表格。你可以从https://www. libreoffice.org/ 和http://www.openoffice.org/ 下载这些软件。即使你的计算机上已经安装了Excel，可能也会发现这些程序更容易使用。但是，本章中的截屏图都来自于Windows 7上的Excel 2010。

12.1 Excel文档

首先，让我们来看一些基本定义。一个Excel电子表格文档称为一个工作簿。一个工作簿保存在扩展名为.xlsx的文件中。每个工作簿可以包含多个表（也称为工作表）。用户当前查看的表（或关闭Excel前最后查看的表），称为活动表。

每个表都有一些列（地址是从A开始的字母）和一些行（地址是从1开始的数字）。在特定行和列的方格称为单元格。每个单元格都包含一个数字或文本值。单元格形成的网格和数据构成了表。

12.2 安装openpyxl模块

Python没有自带openpyxl，所以必须安装。按照附录A中安装第三方模块的指令，模块的名称是 openpyxl。要测试它是否安装正确，就在交互式环境中输入以下代码：

```
>>> import openpyxl
```

如果该模块正确安装，这应该不会产生错误消息。记得在运行本章的交互式环境例子之前，要导入 openpyxl 模块，否则会得到错误，NameError: name 'openpyxl'is not defined。

本书介绍了openpyxl的2.1.4版，但OpenPyXL团队会经常发布新版本。不过不用担心，新版本应该在相当长的时间内向后兼容，支持本书中使用的指令。如果你有新版本，想看看它提供了什么新功能，可以查看OpenPyXL的完整文档：http://openpyxl.readthedocs.org/ 。

12.3 读取Excel文档

本章的例子将使用一个电子表格example.xlsx，它保存在根文件夹中。你可以自己创建这个电子文档，或从http://nostarch.com/automatestuff/ 下载。图12-1展示了3个默认的表，名为Sheet1、Sheet2和Sheet3，这是Excel自动为新工作簿提供的（不同操作系统和电子表格程序，提供的默认表个数可能会不同）。

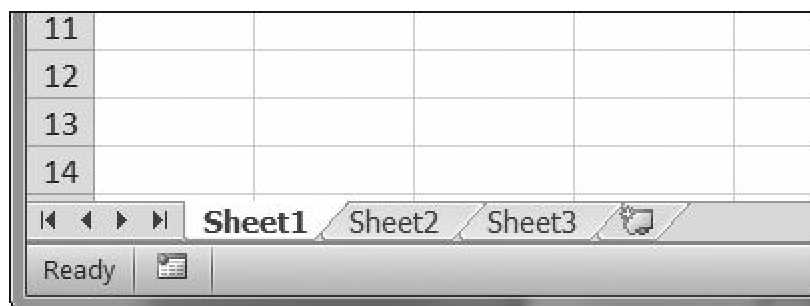


图12-1 工作簿中表的选项卡在Excel的左下角

示例文件中的 Sheet 1 应该看起来像表 12-1（如果你没有从网站下载example.xlsx，就应该在工作表中自己输入这些数据）。

表**12-1 example.xlsx**电子表格

	A	**B**	**C**
1	4/5/2015 1:34:02 PM	Apples	73
2	4/5/2015 3:41:23 AM	Cherries	85
3	4/6/2015 12:46:51 PM	Pears	14
4	4/8/2015 8:59:43 AM	Oranges	52
5	4/10/2015 2:07:00 AM	Apples	152
6	4/10/2015 6:10:37 PM	Bananas	23
7	4/10/2015 2:40:46 AM	Strawberries	98

既然有了示例电子表格，就来看看如何用openpyxl模块来操作它。

12.3.1 用openpyxl模块打开Excel文档

在导入openpyxl模块后，就可以使用openpyxl.load_workbook()函数。在交互式环境中输入以下代码：

```
>>> import openpyxl



>>> wb = openpyxl.load_workbook('example.xlsx')
```

```
>>> type(wb)

< class 'openpyxl.workbook.workbook.Workbook'>
```

openpyxl.load_workbook()函数接受文件名，返回一个workbook数据类型的值。这个workbook对象代表这个Excel文件，有点类似File对象代表一个打开的文本文件。

要记住，example.xlsx需要在当前工作目录，你才能处理它。可以导入os，使用函数os.getcwd()弄清楚当前工作目录是什么，并使用os.chdir()改变当前工作目录。

12.3.2 从工作簿中取得工作表

调用get_sheet_names()方法可以取得工作簿中所有表名的列表。在交互式环境中输入以下代码：

```
>>> import openpyxl

>>> wb = openpyxl.load_workbook('example.xlsx')

>>> wb.get_sheet_names()
```

```
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb.get_sheet_by_name('Sheet3')
>>> sheet
< Worksheet "Sheet3">
>>> type(sheet)
< class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
'Sheet3'
>>> anotherSheet = wb.get_active_sheet()
>>> anotherSheet
< Worksheet "Sheet1">
```

每个表由一个Worksheet对象表示，可以通过向工作簿方法get_sheet_by_name()传递表名字符串获得。最后，可以调用Workbook对象的get_active_sheet()方法，取得工作簿的活动表。活动表是工作簿在Excel中打开时出现的工作表。在取得Worksheet对象后，可以通过title属性取得它的名称。

12.3.3 从表中取得单元格

有了Worksheet对象后，就可以按名字访问Cell对象。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet['A1']
< Cell Sheet1.A1>
>>> sheet['A1'].value
datetime.datetime(2015, 4, 5, 13, 34, 2)
>>> c = sheet['B1']
```

```
>>> c.value
'Apples'
>>> 'Row ' + str(c.row) + ', Column ' + c.column + ' is ' + c.value
'Row 1, Column B is Apples'
>>> 'Cell ' + c.coordinate + ' is ' + c.value
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

Cell对象有一个value属性，不出意外，它包含这个单元格中保存的值。Cell对象也有row、column和coordinate属性，提供该单元格的位置信息。

这里，访问单元格B1的Cell对象的value属性，我们得到字符串'Apples'。row属性给出的是整数1，column属性给出的是'B'，coordinate属性给出的是'B1'。

openpyxl模块将自动解释列A中的日期，将它们返回为datetime值，而不是字符串。datetime数据类型将在第16章中进一步解释。

用字母来指定列，这在程序中可能有点奇怪，特别是在Z列之后，列开时使用两个字母：AA、AB、AC等。作为替代，在调用表的cell()方法时，可以传入整数作为row和column关键字参数，也可以得到一个单元格。第一行或第一列的整数是1，不是0。输入以下代码，继续交互式环境的例子：

```
>>> sheet.cell(row=1, column=2)
< Cell Sheet1.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2):
        print(i, sheet.cell(row=i, column=2).value)
1 Apples
3 Pears
5 Apples
7 Strawberries
```

可以看到，使用表的cell()方法，传入row=1和column=2，将得到单元格B1的Cell对象，就像指定sheet['B1']一样。然后，利用cell()方法和它的关键字参数，就可以编写for循环，打印出一系列单元格的值。

假定你想顺着B列，打印出所有奇数行单元格的值。通过传入2作为range()函数的“步长”参数，可以取得每隔一行的单元格（在这里就是所有奇数行）。for循环的i变量被传递作为cell()方法的row关键字参数，而column关键字参数总是取2。请注意传入的是整数2，而不是字符串'B'。

可以通过Worksheet对象的get_highest_row()和get_highest_column()方法，确定表的大小。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> sheet.get_highest_row()
7
>>> sheet.get_highest_column()
3
```

请注意，get_highest_column()方法返回一个整数，而不是Excel中出现的字母。

12.3.4 列字母和数字之间的转换

要从字母转换到数字，就调用openpyxl.cell.column_index_from_string()函数。要从数字转换到字母，就调用openpyxl.cell.get_column_letter()函数。在交互式环境中输入以下代码：

```
>>> import openpyxl

>>> from openpyxl.cell import get_column_letter, column_index_from_string

>>> get_column_letter(1)

'A'
>>> get_column_letter(2

)
'B'
>>> get_column_letter(27)
```

```
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.get_sheet_by_name('Sheet1')
>>> get_column_letter(sheet.get_highest_column())
'C'
>>> column_index_from_string('A')
1
>>> column_index_from_string('AA')
27
```

在从openpyxl.cell模块引入这两个函数后，可以调用get_column_letter()，传入像27这样的整数，弄清楚第27列的字母是什么。函数column_index_string()做的事情相反：传入一列的字母名称，它告诉你该列的数字是什么。要使用这些函数，不必加载一个工作簿。如果你愿意，可以加载一个工作簿，取得Worksheet对象，并调用Worksheet对象的方法，如get_highest_column()，来取得一个整数。然后，将该整数传递给get_column_letter()。

12.3.5 从表中取得行和列

可以将Worksheet对象切片，取得电子表格中一行、一列或一个矩形区域中的所有Cell对象。然后可以循环遍历这个切片中的所有单元格。在交互式环境中输入以下代码：

```
>>> import openpyxl

>>> wb = openpyxl.load_workbook('example.xlsx')

>>> sheet = wb.get_sheet_by_name('Sheet1')

>>> tuple(sheet['A1':'C3'])

((< Cell Sheet1.A1>, < Cell Sheet1.B1>, < Cell Sheet1.C1>), (< Cell Sheet
< Cell Sheet1.B2>, < Cell Sheet1.C2>), (< Cell Sheet1.A3>, < Cell Sheet1.
< Cell Sheet1.C3>))
❶ >>> for rowOfCellObjects in sheet['A1':'C3']:
```

```
❷         for cellObj in rowOfCellObjects:
              print(cellObj.coordinate, cellObj.value)
          print('--- END OF ROW ---')
```

```
  A1 2015-04-05 13:34:02
  B1 Apples
  C1 73
  ---
END OF ROW ---
  A2 2015-04-05 03:41:23
  B2 Cherries
  C2 85
  ---
END OF ROW ---
```

```
  A3 2015-04-06 12:46:51
  B3 Pears
  C3 14
  ---
END OF ROW ---
```

这里，我们指明需要从A1到C3的矩形区域中的Cell对象，得到了一个Generator对象，它包含该区域中的Cell对象。为了帮助我们看清楚这个Generator对象，可以使用它的tuple()方法，在一个元组中列出它的Cell对象。

这个元组包含3个元组：每个元组代表1行，从指定区域的顶部到底部。这3个内部元组中的每一个包含指定区域中一行的Cell对象，从最左边的单元格到最右边。所以总的来说，工作表的这个切片包含了从A1到C3区域的所有Cell对象，从左上角的单元格开始，到右下角的单元格结束。

要打印出这个区域中所有单元格的值，我们使用两个for循环。外层for循环遍历这个切片中的每一行❶。然后针对每一行，内层for循环遍历该行中的每个单元格❷。

要访问特定行或列的单元格的值，也可以利用Worksheet对象的rows和columns属性。在交互式环境中输入以下代码：

```
>>> import openpyxl
```

```
>>> wb = openpyxl.load_workbook('example.xlsx')

>>> sheet = wb.get_active_sheet()

>>> sheet.columns[1]

(< Cell Sheet1.B1>, < Cell Sheet1.B2>, < Cell Sheet1.B3>, < Cell Sheet1.B4>
< Cell Sheet1.B5>, < Cell Sheet1.B6>, < Cell Sheet1.B7>)
>>> for cellObj in sheet.columns[1]:

        print(cellObj.value)

Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

利用Worksheet对象的rows属性，可以得到一个元组构成的元组。内部的每个元组都代表1行，包含该行中的Cell对象。columns属性也会给你一个元组构成的元组，内部的每个元组都包含1列中的Cell对象。对于example.xlsx，因为有7行3列，rows给出由7个元组构成的一个元组（每个内部元组包含3个Cell对象）。columns给出由3个元组构成的一个元组（每个内部元组包含7个Cell对象）。

要访问一个特定的元组，可以利用它在大的元组中的下标。例如，要得到代表B列的元组，可以用sheet.columns[1]。要得到代表A列的元组，可以用sheet.columns[0]。在得到了代表行或列的元组后，可以循环遍历它的对象，打印出它们的值。

12.3.6 工作簿、工作表、单元格

作为快速复习，下面是从电子表格文件中读取单元格涉及的所有函数、方法和数据类型。

1．导入openpyxl模块。

2．调用openpyxl.load_workbook()函数。

3．取得Workbook对象。

4．调用get_active_sheet()或get_sheet_by_name()工作簿方法。

5．取得Worksheet对象。

6．使用索引或工作表的cell()方法，带上row和column关键字参数。

7．取得Cell对象。

8．读取Cell对象的value属性。

12.4 项目：从电子表格中读取数据

假定你有一张电子表格的数据，来自于2010年美国人口普查。你有

一个无聊的任务，要遍历表中的几千行，计算总的人口，以及每个县的普查区的数目（普查区就是一个地理区域，是为人口普查而定义的）。每行表示一个人口普查区。我们将这个电子表格文件命名为censuspopdata.xlsx，可以从http://nostarch.com/automatestuff/下载它。它的内容如图12-2所示。

尽管Excel是要能够计算多个选中单元格的和，你仍然需要选中3000个以上县的单元格。即使手工计算一个县的人口只需要几秒钟，整张电子表格也需要几个小时时间。

	A	B	C	D	E
1	CensusTract	State	County	POP2010	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	

Population by Census Tract

Ready

图12-2 censuspopdata.xlsx电子表格

在这个项目中，你要编写一个脚本，从人口普查电子表格文件中读取数据，并在几秒钟内计算出每个县的统计值。

下面是程序要做的事：

- 从Excel电子表格中读取数据。
- 计算每个县中普查区的数目。
- 计算每个县的总人口。
- 打印结果。

这意味着代码需要完成下列任务：

- 用openpyxl模块打开Excel文档并读取单元格。
- 计算所有普查区和人口数据，将它保存到一个数据结构中。

- 利用pprint模块，将该数据结构写入一个扩展名为.py的文本文件。

第1步：读取电子表格数据

censuspopdata.xlsx电子表格中只有一张表，名为'Population by Census Tract'。每一行都保存了一个普查区的数据。列分别是普查区的编号（A），州的简称（B），县的名称（C），普查区的人口（D）。

打开一个新的文件编辑器窗口，输入以下代码。将文件保存为readCensusExcel.py。

```
  #! python3
  # readCensusExcel.py - Tabulates population and number of census tracts f
  # each county.

❶ import openpyxl, pprint
  print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb.get_sheet_by_name('Population by Census Tract')
  countyData = {}

  # TODO: Fill in countyData with each county's population and tracts.
  print('Reading rows...')
❹ for row in range(2, sheet.get_highest_row() + 1):
      # Each row in the spreadsheet has data for one census tract.
      State  = sheet['B' + str(row)].value
      county = sheet['C' + str(row)].value
      pop    = sheet['D' + str(row)].value

  # TODO: Open a new text file and write the contents of countyData to it.
```

这段代码导入了openpyxl模块，也导入了pprint模块，你用后者来打印最终的县的数据❶。然后代码打开了censuspopdata.xlsx文件❷，取得了包含人口普查数据的工作表❸，开始迭代它的行❹。

请注意，你也创建了一个countyData变量，它将包含你计算的每个县的人口和普查区数目。但在它里面存储任何东西之前，你应该确定它内部的数据结构。

第2步：填充数据结构

保存在countyData中的数据结构将是一个字典，以州的简称作为键。每个州的简称将映射到另一个字典，其中的键是该州的县的名称。每个县的名称又映射到一个字典，该字典只有两个键，'tracts'和'pop'。这些键映射到普查区数目和该县的人口。例如，该字典可能类似于：

```
{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},
        'Aleutians West': {'pop': 5561, 'tracts': 2},
        'Anchorage': {'pop': 291826, 'tracts': 55},
        'Bethel': {'pop': 17013, 'tracts': 3},
        'Bristol Bay': {'pop': 997, 'tracts': 1},
        --snip



--
```

如果前面的字典保存在countyData中，下面的表达式求值结果如下：

```
>>> countyData['AK']['Anchorage']['pop']
291826
>>> countyData['AK']['Anchorage']['tracts']
55
```

一般来说，countyData字典中的键看起来像这样：

```
countyData[state abbrev




][county
```

```
][ 'tracts' ]
countyData[state abbrev

][county

][ 'pop' ]
```

既然知道了countyData的结构，就可以编写代码，用县的数据填充它。将下面的代码添加到程序的末尾：

```
  #! python 3
  # readCensusExcel.py - Tabulates population and number of census tracts f
  # each county.

  --snip

--
  for row in range(2, sheet.get_highest_row() + 1):
      # Each row in the spreadsheet has data for one census tract.
      State  = sheet['B' + str(row)].value
      county = sheet['C' + str(row)].value
      pop     = sheet['D' + str(row)].value

      # Make sure the key for this state exists.
```

```
❶     countyData.setdefault(state, {})

      # Make sure the key for this county in this state exists.

❷     countyData[state].setdefault(county, {'tracts': 0, 'pop': 0})



      # Each row represents one census tract, so increment by one.

❸     countyData[state][county]['tracts'] += 1

      # Increase the county pop by the pop in this census tract.

❹     countyData[state][county]['pop'] += int(pop)
```

```
# TODO: Open a new text file and write the contents of countyData to it.
```

最后的两行代码执行实际的计算工作，在for循环的每次迭代中，针对当前的县，增加tracts的值❸，并增加pop的值❹。

其他代码存在是因为，只有countyData中存在的键，你才能引用它的值。（也就是说，如果'AK'键不存在，countyData['AK']['Anchorage']['tracts'] += 1将导致一个错误）。为了确保州简称的键存在，你需要调用setdefault()方法，在state还不存在时设置一个默认值❶。

正如countyData字典需要一个字典作为每个州缩写的值，这样的字典又需要一个字典，作为每个县的键的值❷。这样的每个字典又需要键'tracts'和'pop'，它们的初始值为整数0（如果这个字典的结构令你混淆，回去看看本节开始处字典的例子）。

如果键已经存在，setdefault()不会做任何事情，因此在for循环的每次迭代中调用它不会有问题。

第3步：将结果写入文件

for循环结束后，countyData字典将包含所有的人口和普查区信息，以县和州为键。这时，你可以编写更多代码，将数据写入文本文件或另一个Excel电子表格。目前，我们只是使用pprint.pformat()函数，将变量字典的值作为一个巨大的字符串，写入文件census2010.py。在程序的末尾加上以下代码（确保它没有缩进，这样它就在for循环之外）：

```
#! python 3
# readCensusExcel.py - Tabulates population and number of census tracts for
# each county.
--snip



--
```

```
for row in range(2, sheet.get_highest_row() + 1):
--snip

--

# Open a new text file and write the contents of countyData to it.

print('Writing results...')

resultFile = open('census2010.py', 'w')

resultFile.write('allData = ' + pprint.pformat(countyData))

resultFile.close()

print('Done.')
```

pprint.pformat()函数产生一个字符串，它本身就是格式化好的、有效的Python代码。将它输出到文本文件census2010.py，你就通过Python程序生成了一个Python程序！这可能看起来有点复杂，但好处是你现在可以导入census2010.py，就像任何其他Python模块一样。在交互式环境中，将当前工作目录变更到新创建的文件所在的文件夹（在我的笔记本上，就是C:\Python34），然后导入它：

```
>>> import os

>>> os.chdir('C:\\Python34')

>>> import census2010

>>> census2010.allData['AK']['Anchorage']

{'pop': 291826, 'tracts': 55}
>>> anchoragePop = census2010.allData['AK']['Anchorage']['pop']

>>> print('The 2010 population of Anchorage was ' + str(anchoragePop))
```

```
The 2010 population of Anchorage was 291826
```

readCensusExcel.py程序是可以扔掉的代码：当你把它的结果保存为census2010.py之后，就不需要再次运行该程序了。任何时候，只要需要县的数据，就可以执行import census2010。

手工计算这些数据可能需要数小时，这个程序只要几秒钟。利用OpenPyXL，可以毫无困难地提取保存在 Excel 电子表格中的信息，并对它进行计算。从http://nostarch. com/automatestuff/可以下载这个完整的程序。

第4步：类似程序的思想

许多公司和组织机构使用Excel来保存各种类型的数据，电子表格会变得庞大，这并不少见。解析Excel电子表格的程序都有类似的结构：它加载电子表格文件，准备一些变量或数据结构，然后循环遍历电子表格中的每一行。这样的程序可以做下列事情：

- 比较一个电子表格中多行的数据。
- 打开多个Excel文件，跨电子表格比较数据。
- 检查电子表格是否有空行或无效的数据，如果有就警告。
- 从电子表格中读取数据，将它作为Python程序的输入。

12.5 写入Excel文档

OpenPyXL也提供了一些方法写入数据，这意味着你的程序可以创建和编辑电子表格文件。利用Python，创建一个包含几千行数据的电子表格是非常简单的。

12.5.1 创建并保存Excel文档

调用openpyxl.Workbook()函数，创建一个新的空Workbook对象。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> sheet = wb.get_active_sheet()
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'
>>> wb.get_sheet_names()
['Spam Bacon Eggs Sheet']
```

工作簿将从一个工作表开始，名为Sheet。你可以将新的字符串保存在它的title属性中，从而改变工作表的名字。

当修改Workbook对象或它的工作表和单元格时，电子表格文件不会保存，除非你调用save()工作簿方法。在交互式环境中输入以下代码（让example.xlsx处于当前工作目录）：

```
>>> import openpyxl

>>> wb = openpyxl.load_workbook('example.xlsx')

>>> sheet = wb.get_active_sheet()

>>> sheet.title = 'Spam Spam Spam'

>>> wb.save('example_copy.xlsx')
```

这里，我们改变了工作表的名称。为了保存变更，我们将文件名作为字符串传递给save()方法。传入的文件名与最初的文件名不同，例如'example_copy.xlsx'，这将变更保存到电子表格的一份拷贝中。

当你编辑从文件中加载的一个电子表格时，总是应该将新的、编辑过的电子表格保存到不同的文件名中。这样，如果代码中有缺陷，导致新的保存到文件中数据不对或讹误，还有最初的电子表格文件可以处理。

12.5.2 创建和删除工作表

利用create_sheet() and remove_sheet()方法，可以在工作簿中添加或删除工作表。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.get_sheet_names()
['Sheet']
>>> wb.create_sheet()
<  Worksheet "Sheet1">
>>> wb.get_sheet_names()
```

```
['Sheet', 'Sheet1']
>>> wb.create_sheet(index=0, title='First Sheet')
<  Worksheet "First Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<  Worksheet "Middle Sheet">
>>> wb.get_sheet_names()
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

create_sheet()方法返回一个新的Worksheet对象，名为SheetX，它默认是工作簿的最后一个工作表。或者，可以利用index和title关键字参数，指定新工作表的索引或名称。

继续前面的例子，输入以下代码：

```
>>> wb.get_sheet_names()
```

```
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> wb.remove_sheet(wb.get_sheet_by_name('Middle Sheet'))
>>> wb.remove_sheet(wb.get_sheet_by_name('Sheet1'))
>>> wb.get_sheet_names()
['First Sheet', 'Sheet']
```

remove_sheet()方法接受一个Worksheet对象作为其参数，而不是工作表名称的字符串。如果你只知道要删除的工作表的名称，就调用get_sheet_by_name()，将它的返回值传入remove_sheet()。

在工作簿中添加或删除工作表之后，记得调用save()方法来保存变更。

12.5.3 将值写入单元格

将值写入单元格，很像将值写入字典中的键。在交互式环境中输入以下代码：

```
>>> import openpyxl
```

```
>>> wb = openpyxl.Workbook()

>>> sheet = wb.get_sheet_by_name('Sheet')

>>> sheet['A1'] = 'Hello world!'

>>> sheet['A1'].value

'Hello world!'
```

如果你有单元格坐标的字符串，可以像字典的键一样，将它用于Worksheet对象，指定要写入的单元格。

12.6 项目：更新一个电子表格

这个项目需要编写一个程序，更新产品销售电子表格中的单元格。程序将遍历这个电子表格，找到特定类型的产品，并更新它们的价格。请从http://nostarch.com/ automatestuff/下载这个电子表格。图12-3展示了这个电子表格。

	A	B	C	D	E
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL	
2	Potatoes	0.86	21.6	18.58	
3	Okra	2.26	38.6	87.24	
4	Fava beans	2.69	32.8	88.23	
5	Watermelon	0.66	27.3	18.02	
6	Garlic	1.19	4.9	5.83	
7	Parsnips	2.27	1.1	2.5	
8	Asparagus	2.49	37.9	94.37	
9	Avocados	3.23	9.2	29.72	
10	Celery	3.07	28.9	88.72	
11	Okra	2.26	40	90.4	

Sheet

Ready 100%

图12-3 产品销售的电子表格

每一行代表一次单独的销售。列分别是销售产品的类型（A）、产品每磅的价格（B）、销售的磅数（C），以及这次销售的总收入。TOTAL列设置为Excel公式，将每磅的成本乘以销售的磅数，并将结果取整到分。有了这个公式，如果列B或C发生变化，TOTAL列中的单元格将自动更新。

现在假设Garlic、Celery和Lemons的价格输入的不正确。这让你面对一项无聊的任务：遍历这个电子表格中的几千行，更新所有garlic、celery和lemon行中每磅的价格。你不能简单地对价格查找替换，因为可能有其他的产品价格一样，你不希望错误地"更正"。对于几千行数据，手工操作可能要几小时。但你可以编写程序，几秒钟内完成这个任务。

你的程序做下面的事情：

- 循环遍历所有行。
- 如果该行是Garlic、Celery或Lemons，更新价格。

这意味着代码需要做下面的事情：

- 打开电子表格文件。
- 针对每一行，检查列A的值是不是Celery、Garlic或Lemon。
- 如果是，更新列B中的价格。

- 将该电子表格保存为一个新文件（这样就不会丢失原来的电子表格，以防万一）。

第**1**步：利用更新信息建立数据结构

需要更新的价格如下：

Celery 1.19

Garlic 3.07

Lemon 1.27

你可以像这样编写代码：

```
if produceName == 'Celery':
    cellObj = 1.19
if produceName == 'Garlic':
    cellObj = 3.07
if produceName == 'Lemon':
    cellObj = 1.27
```

这样硬编码产品和更新的价格有点不优雅。如果你需要用不同的价格，或针对不同的产品，再次更新这个电子表格，就必须修改很多代码。每次修改代码，都有引入缺陷的风险。

更灵活的解决方案，是将正确的价格信息保存在字典中，在编写代码时，利用这个数据结构。在一个新的文件编辑器窗口中，输入以下代码：

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet.

import openpyxl

wb = openpyxl.load_workbook('produceSales.xlsx')
sheet = wb.get_sheet_by_name('Sheet')
```

```
# The produce types and their updated prices
PRICE_UPDATES = {'Garlic': 3.07,
                 'Celery': 1.19,
                 'Lemon': 1.27}

# TODO: Loop through the rows and update the prices.
```

将它保存为updateProduce.py。如果需要再次更新这个电子表格，只需要更新PRICE_UPDATES字典，不用修改其他代码。

第2步：检查所有行，更新不正确的价格

程序的下一部分将循环遍历电子表格中的所有行。将下面代码添加到updateProduce.py的末尾：

```
  #! python3
  # updateProduce.py - Corrects costs in produce sales spreadsheet.

  --snip

--

  # Loop through the rows and update the prices.

❶ for rowNum in range(2, sheet.get_highest_row()):    # skip the first row

❷     produceName = sheet.cell(row=rowNum, column=1).value
```

```
❸     if produceName in PRICE_UPDATES:

          sheet.cell(row=rowNum, column=2).value = PRICE_UPDATES[produceNam

❹ wb.save('updatedProduceSales.xlsx')
```

我们从第二行开始循环遍历，因为第1行是标题❶。第1列的单元格（即列A）将保存在变量produceName中❷。如果produceName的值是PRICE_ UPDATES字典中的一个键❸，你就知道，这行的价格必须修改。正确的价格是PRICE_UPDATES[produceName]。

请注意，使用PRICE_UPDATES让代码变得多么干净。只需要一条if语句，而不是像if produceName == 'Garlic'这样的代码，就能够更新所有类型的产品。因为代码没有硬编码产品名称，而是使用PRICE_UPDATES字典，在for循环中更新价格，所以如果产品销售电子表格需要进一步修改，你只需要修改PRICE_UPDATES字典，不用改其他代码。

在遍历整个电子表格并进行修改后，代码将Workbook对象保存到updatedProduceSales. xlsx❹。它没有覆写原来的电子表格，以防万一程序有缺陷，将电子表格改错。在确认修改的电子表格正确后，你可以删除原来的电子表格。

你可以从http://nostarch.com/automatestuff/下载这个程序的完整源代码。

第3步：类似程序的思想

因为许多办公室职员一直在使用Excel电子表格，所以能够自动编辑和写入Excel文件的程序，将非常有用。这样的程序可以完成下列任务：

- 从一个电子表格读取数据，写入其他电子表格的某些部分。
- 从网站、文本文件或剪贴板读取数据，将它写入电子表格。
- 自动清理电子表格中的数据。例如，可以利用正则表达式，读取多种格式的电话号码，将它们转换成单一的标准格式。

12.7 设置单元格的字体风格

设置某些单元格行或列的字体风格，可以帮助你强调电子表格中重点的区域。例如，在这个产品电子表格中，程序可以对potato、garlic和parsnip等行使用粗体。或者也许你希望对每磅价格超过5美元的行使用斜体。手工为大型电子表格的某些部分设置字体风格非常令人厌烦，但程序可以马上完成。

为了定义单元格的字体风格，需要从openpyxl.styles模块导入Font()和Style()函数。

```
from openpyxl.styles import Font, Style
```

这让你能输入Font()，代替openpyxl.styles.Font()（参见2.8节“导入模块”，复习这种方式的import语句）。

这里有一个例子，它创建了一个新的工作簿，将A1单元格设置为24点、斜体。在交互式环境中输入以下代码：

```
>>> import openpyxl
```

```
   >>> from openpyxl.styles import Font, Style
   >>> wb = openpyxl.Workbook()
   >>> sheet = wb.get_sheet_by_name('Sheet')
❶ >>> italic24Font = Font(size=24, italic=True)
❷ >>> styleObj = Style(font=italic24Font)
❸ >>> sheet['A'].style/styleObj
   >>> sheet['A1'] = 'Hello world!'
   >>> wb.save('styled.xlsx')
```

OpenPyXL模块用Style对象来表示单元格字体风格设置的集合，字体风格保存在Cell对象的style属性中。将Style对象赋给style属性，可以设置单元格的字体风格。

在这个例子中，Font(size=24, italic=True)返回一个 Font 对象，保存在italic24Font中❶。Font()的关键字参数size和italic，配置了Font对象的style属性。这个Font对象被传递给Style(font=italic24Font)调用，该函数的返回值保存在styleObj中❷。如果styleObj被赋给单元格的style属性❸，所有字体风格的信息将应用于单元格A1。

12.8 Font对象

Font对象的style属性影响文本在单元格中的显示方式。要设置字体风格属性，就向Font()函数传入关键字参数。表12-2展示了Font()函数可能的关键字参数。

表**12-2 Font style**属性的关键字参数

关键字参数	数据类型	描述
name	字符串	字体名称，诸如'Calibri' 或'Times New Roman'
size	整型	大小点数
bold	布尔型	True表示粗体
italic	布尔型	True表示斜体

可以调用Font()来创建一个Font对象，并将这个Font对象保存在一个变量中。然后将它传递给Style()，得到的Style对象保存在一个变量中，并将该变量赋给Cell对象的style属性。例如，下面的代码创建了各种字体风格：

```
>>> import openpyxl

>>> from openpyxl.styles import Font, Style

>>> wb = openpyxl.Workbook()

>>> sheet = wb.get_sheet_by_name('Sheet')

>>> fontObj1 = Font(name='Times New Roman', bold=True)

>>> styleObj1 = Style(font=fontObj1)

>>> sheet['A1'].style/styleObj
```

```
>>> sheet['A1'] = 'Bold Times New Roman'

>>> fontObj2 = Font(size=24, italic=True)

>>> styleObj2 = Style(font=fontObj2)

>>> sheet['B3'].style/styleObj

>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles.xlsx')
```

这里，我们将一个Font对象保存在fontObj1中，并用它创建一个Style对象，该对象保存在styleObj1中，然后将A1的Cell对象的style属性设置为styleObj。我们针对另一个Font对象和Style对象重复这个过程，设置第二个单元格的字体风格。运行这段代码后，电子表格中A1和B3单元格的字体风格将设置为自定义的字体风格，如图12-4所示。

	A	B	C	D
1	**Bold Times New Roman**			
2				
3		*24 pt Italic*		
4				
5				

图12-4 带有自定义字体风格的电子表格

对于单元格A1，我们将字体名称设置为'Times New Roman'，并将bold设置为true，这样我们的文本将以粗体Times New Roman的方式显示。我们没有指定大小，所以使用openpyxl的默认值11。在单元格B3中，我们的文本是斜体，大小是24。我们没有指定字体的名称，所以使用openpyxl的默认值Calibri。

12.9 公式

公式以一个等号开始，可以配置单元格，让它包含通过其他单元格计算得到的值。在本节中，你将利用openpyxl模块，用编程的方式在单元格中添加公式，就像添加普通的值一样。例如：

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

这将=SUM(B1:B8)作为单元格B9的值。这将B9单元格设置为一个公式，计算单元格B1到B8的和。图12-5展示了它的效果。

Clipboard　Font　Alignment　Nu

B9　=SUM(B1:B8)

	A	B	C	D	E
1		82			
2		11			
3		85			
4		18			
5		57			
6		51			
7		38			
8		42			
9	**TOTAL:**	**384**			
10					

Sheet1　Sheet2　Sheet3

Ready

图12-5 单元格B9包含了一个公式，计算单元格B1到B8的和

为单元格设置公式就像设置其他文本值一样。在交互式环境中输入以下代码：

```
>>> import openpyxl

>>> wb = openpyxl.Workbook()

>>> sheet = wb.get_active_sheet()
```

```
>>> sheet['A1'] = 200

>>> sheet['A2'] = 300

>>> sheet['A3'] = '=SUM(A1:A2)'

>>> wb.save('writeFormula.xlsx')
```

单元格A1和A2分别设置为200和300。单元格A3设置为一个公式，求出A1和A2的和。如果在Excel中打开这个电子表格，A3的值将显示为500。

也可以读取单元格中的公式，就像其他值一样。但是，如果你希望看到该公式的计算结果，而不是原来的公式，就必须将load_workbook()的data_only关键字参数设置为True。这意味着Workbook对象要么显示公式，要么显示公式的结果，不能兼得（但是针对一个电子表格文件，可以加载多个Workbook对象）。在交互式环境中输入以下代码，看看有无data_only关键字参数时，加载工作簿的区别：

```
>>> import openpyxl
```

```
>>> wbFormulas = openpyxl.load_workbook('writeFormula.xlsx')

>>> sheet = wbFormulas.get_active_sheet()

>>> sheet['A3'].value

'=SUM(A1:A2)'
>>> wbDataOnly = openpyxl.load_workbook('writeFormula.xlsx', data_only=True

>>> sheet = wbDataOnly.get_active_sheet()

>>> sheet['A3'].value

500
```

这里，如果调用load_workbook()时带有data_only=True，A3单元格就显示为500，即公式的结果，而不是公式的文本。

Excel公式为电子表格提供了一定程度的编程能力，但对于复杂的任务，很快就会失去控制。例如，即使你非常熟悉Excel的公式，要想弄清楚=IFERROR
(TRIM(IF(LEN(VLOOKUP(F7,Sheet2!$A$1:$B$10000,2,FALSE))>0,SUBS
(VLOOKUP (F7, Sheet2!$A$1:$B$10000, 2, FALSE), " ", ""),"")), "")实际上做了什么，也是一件非常头痛的事。Python代码的可读性要好得多。

12.10 调整行和列

在Excel中，调整行和列的大小非常容易，只要点击并拖动行的边缘，或列的头部。但如果你需要根据单元格的内容来设置行或列的大小，或者希望设置大量电子表格文件中的行列大小，编写Python程序来做就要快得多。

行和列也可以完全隐藏起来。或者它们可以“冻结”，这样就总是显示在屏幕上，如果打印该电子表格，它们就出现在每一页上（这很适合做表头）。

12.10.1 设置行高和列宽

Worksheet对象有row_dimensions和column_dimensions属性，控制行高和列宽。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.get_active_sheet()
```

```
>>> sheet['A1'] = 'Tall row'

>>> sheet['B2'] = 'Wide column'

>>> sheet.row_dimensions[1].height = 70

>>> sheet.column_dimensions['B'].width = 20

>>> wb.save('dimensions.xlsx')
```

工作表的row_ dimensions和column_ dimensions是像字典一样的值，row_ dimensions包含RowDimension对象，column_ dimensions包含ColumnDimension对象。在row_dimensions中，可以用行的编号来访问一个对象（在这个例子中，是1或）。在column_ dimensions中，可以用列的字母来访问一个对象（在这个例子中，是A或B）。

dimensions.xlsx电子表格如图12-6所示。

	A	B	
1	Tall row		
2		Wide column	
3			

图12-6 行1和列B设置了更大的高度和宽度

一旦有了RowDimension对象，就可以设置它的高度。一旦有了ColumnDimension对象，就可以设置它的宽度。行的高度可以设置为0到409之间的整数或浮点值。这个值表示高度的点数。一点等于1/72英寸。默认的行高是12.75。列宽可以设置为0到255之间的整数或浮点数。这个值表示使用默认字体大小时（11点），单元格可以显示的字符数。默认的列宽是8.43个字符。列宽为零或行高为零，将使单元格隐藏。

12.10.2 合并和拆分单元格

利用merge_cells()工作表方法，可以将一个矩形区域中的单元格合并为一个单元格。在交互式环境中输入以下代码：

```
>>> import openpyxl

>>> wb = openpyxl.Workbook()

>>> sheet = wb.get_active_sheet()
```

```
>>> sheet.merge_cells('A1:D3')
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5')
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged.xlsx')
```

merge_cells()的参数是一个字符串，表示要合并的矩形区域左上角和右下角的单元格：'A1:D3'将12个单元格合并为一个单元格。要设置这些合并后单元格的值，只要设置这一组合并单元格左上角的单元格的值。

如果运行这段代码，merged.xlsx看起来如图12-7所示。

	A	B	C	D	E
1					
2					
3	12个单元格合并到一起				
4					
5			两个单元格合并到一起		
6					
7					

图12-7 在电子表格中合并单元格

要拆分单元格，就调用 unmerge_cells()工作表方法。在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.unmerge_cells('A1:D3')
>>> sheet.unmerge_cells('C5:D5')
```

```
>>> wb.save('merged.xlsx')
```

如果保存变更，然后查看这个电子表格，就会看到合并的单元格恢复成一些独立的单元格。

12.10.3 冻结窗格

对于太大而不能一屏显示的电子表格，“冻结”顶部的几行或最左边的几列，是很有帮助的。例如，冻结的列或行表头，就算用户滚动电子表格，也是始终可见的。这称为“冻结窗格”。在OpenPyXL中，每个Worksheet对象都有一个freeze_panes属性，可以设置为一个Cell对象或一个单元格坐标的字符串。请注意，单元格上边的所有行和左边的所有列都会冻结，但单元格所在的行和列不会冻结。

要解冻所有的单元格，就将freeze_panes设置为None或'A1'。表12-3展示了freeze_panes设定的一些例子，以及哪些行或列会冻结。

表**12-3** 冻结窗格的例子

freeze_panes的设置	冻结的行和列
sheet.freeze_panes = 'A2'	行1
sheet.freeze_panes = 'B1'	列A
sheet.freeze_panes = 'C1'	列A和列B
sheet.freeze_panes = 'C2'	行1和列A和列B

sheet.freeze_panes = 'A1'或 sheet.freeze_panes = None	没有冻结窗格

确保你有来自http://nostarch.com/automatestuff/ 的产品销售电子表格。然后在交互式环境中输入以下代码：

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales.xlsx')
>>> sheet = wb.get_active_sheet()
>>> sheet.freeze_panes = 'A2'
>>> wb.save('freezeExample.xlsx')
```

如果将freeze_panes属性设置为'A2'，行1将永远可见，无论用户将电子表格滚动到何处，如图12-8所示。

	A	B	C	D	E	F
1	FRUIT	COST PER POUND	POUNDS SOLD	TOTAL		
1591	Fava beans	2.69	0.7	1.88		
1592	Grapefruit	0.76	28.5	21.66		
1593	Green peppers	1.89	37	69.93		
1594	Watermelon	0.66	30.4	20.06		
1595	Celery	3.07	36.6	112.36		
1596	Strawberries	4.4	5.5	24.2		
1597	Green beans	2.52	40	100.8		

图12-8 将reeze_panes设置为'A2'，行1将永远可见，无论用户如何向下滚动

12.10.4 图表

openpyxl支持利用工作表中单元格的数据，创建条形图、折线图、散点图和饼图。要创建图表，需要做下列事情：

1．从一个矩形区域选择的单元格，创建一个Reference对象。

2．通过传入Reference对象，创建一个Series对象。

3．创建一个Chart对象。

4．将Series对象添加到Chart对象。

5．可选地设置Chart对象的drawing.top、drawing.left、drawing.width和drawing.height变量。

6．将Chart对象添加到Worksheet对象。

Reference对象需要一些解释。Reference对象是通过调用openpyxl.charts. Reference()函数并传入3个参数创建的：

1．包含图表数据的Worksheet对象。

2．两个整数的元组，代表矩形选择区域的左上角单元格，该区域包含图表数据：元组中第一个整数是行，第二个整数是列。请注意第一行是1，不是0。

3．两个整数的元组，代表矩形选择区域的右下角单元格，该区域包含图表数据：元组中第一个整数是行，第二个整数是列。

图12-9展示了坐标参数的一些例子。

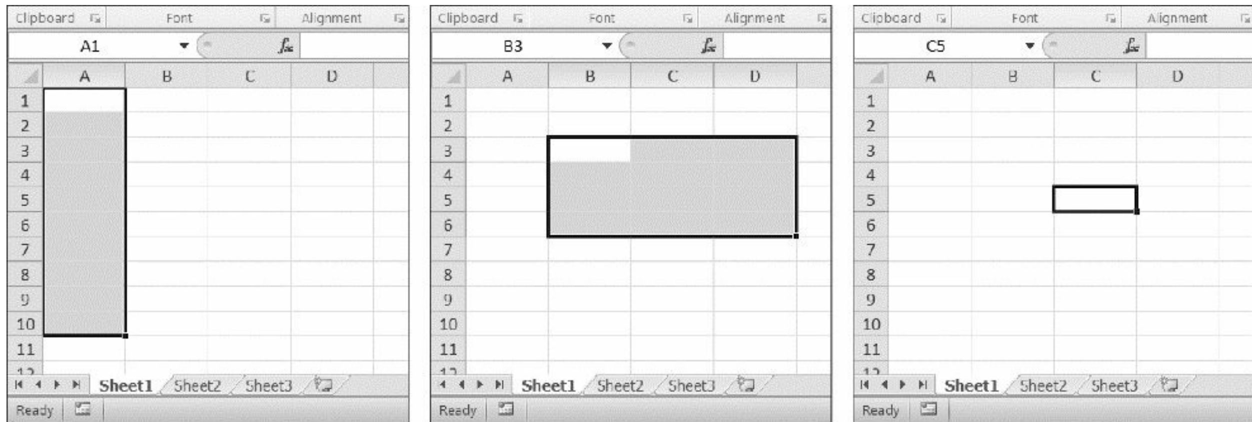


图12-9 从左到右：(1, 1), (10, 1); (3, 2), (6, 4); (5, 3), (5, 3)

在交互式环境中输入以下代码，创建一个条形图，将它添加到电子表格中：

```
>>> import openpyxl

>>> wb = openpyxl.Workbook()

>>> sheet = wb.get_active_sheet()

>>> for i in range(1, 11):          # create some data in column A
```

```
        sheet['A' + str(i)] = i

>>> refObj = openpyxl.charts.Reference(sheet, (1, 1), (10, 1))

>>> seriesObj = openpyxl.charts.Series(refObj, title='First series')

>>> chartObj = openpyxl.charts.BarChart()

>>> chartObj.append(seriesObj)

>>> chartObj.drawing.top = 50         # set the position

>>> chartObj.drawing.left = 100

>>> chartObj.drawing.width = 300     # set the size
```

```
>>> chartObj.drawing.height = 200

>>> sheet.add_chart(chartObj)

>>> wb.save('sampleChart.xlsx')
```

得到的电子表格，如图12-10所示。

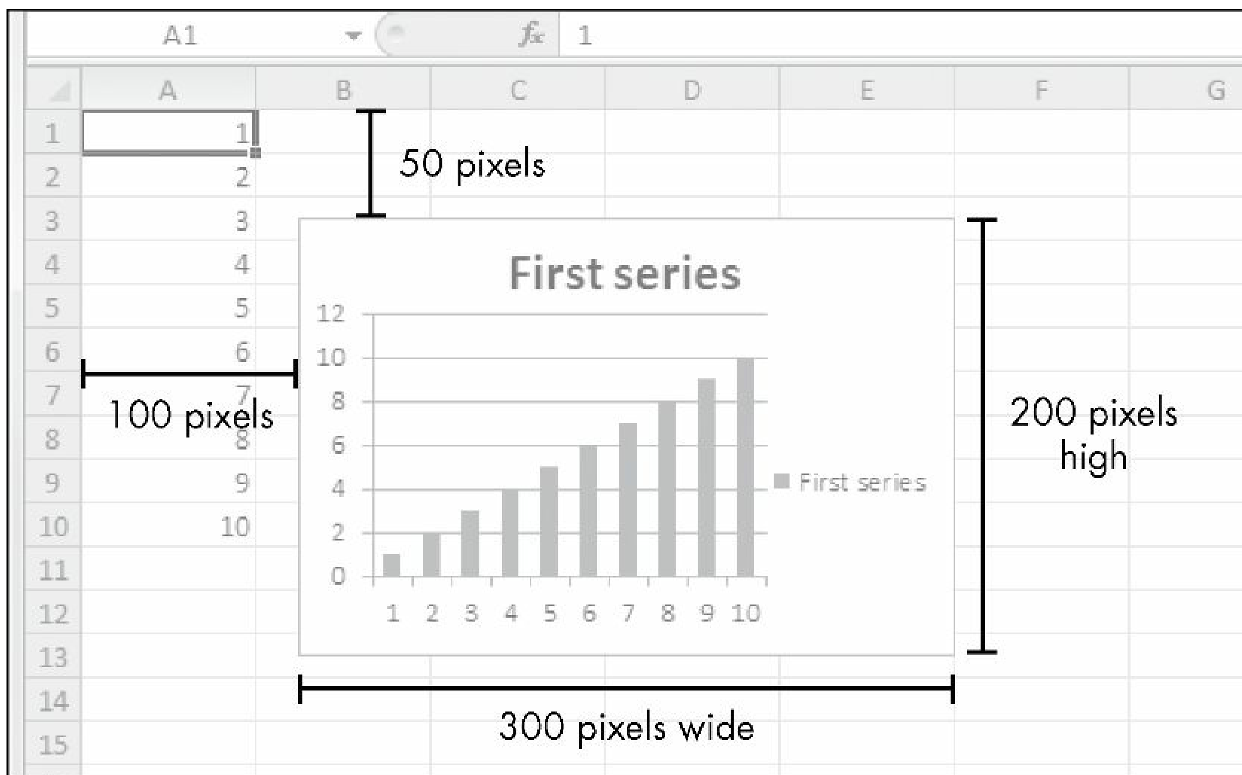


图12-10 添加了一个图表的电子表格

我们可以调用 openpyxl.charts.BarChart()，创建一个条形图。也可以调用openpyxl.charts.LineChart()、openpyxl.charts.ScatterChart()和openpyxl.charts.PieChart()，创建折线图、散点图和饼图。

遗憾的是，在OpenPyXL的当前版本中（2.1.4），load_workbook()不会加载Excel文件中的图表。即使Excel文件包含图表，加载的Workbook对象也不会包含它们。如果加载一个Workbook对象，然后马上保存到同样的.xlsx文件名中，实际上就会删除其中的图表。

12.11 小结

处理信息是比较难的部分，通常不是处理本身难，而是为程序得到正确格式的数据较难。一旦你将电子表格载入Python，就可以提取并操作它的数据，比手工操作要快得多。

你也可以生成电子表格，作为程序的输出。所以如果同事需要将包含几千条销售合同的文本文件或PDF转换成电子表格文件，你就不需要无聊地将它拷贝粘贴到Excel中。

有了openpyxl模块和一些编程知识，你会发现处理很大的电子表格也是小事一桩。

12.12 习题

对于以下的问题，设想你有一个Workbook对象保存在变量wb中，一个Worksheet对象保存在sheet中，一个Cell对象保存在cell中，一个Comment对象保存在comm中，一个Image对象保存在img中。

1．openpyxl.load_workbook()函数返回什么？

2．get_sheet_names()工作簿方法返回什么？

3．如何取得名为'Sheet1'的工作表的Worksheet对象？

4．如何取得工作簿的活动工作表的Worksheet对象？

5．如何取得单元格C5中的值？

6．如何将单元格C5中的值设置为"Hello"？

7．如何取得表示单元格的行和列的整数？

8．工作表方法get_highest_column()和get_highest_row()返回什么？这些返回值的类型是什么？

9．如果要取得列'M'的整数下标，需要调用什么函数？

10．如果要取得列14的字符串名称，需要调用什么函数？

11．如何取得从A1到F1的所有Cell对象的元组？

12．如何将工作簿保存到文件名example.xlsx？

13．如何在一个单元格中设置公式？

14．如果需要取得单元格中公式的结果，而不是公式本身，必须先做什么？

15．如何将第5行的高度设置为100？

16．如何设置列C的宽度？

17．列出一些openpyxl 2.1.4不会从电子表格文件中加载的功能。

18．什么是冻结窗格？

19．创建一个条形图，需要调用哪5个函数和方法？

12.13 实践项目

作为实践，编程执行以下任务。

12.13.1 乘法表

创建程序multiplicationTable.py，从命令行接受数字N，在一个Excel电子表格中创建一个N×N的乘法表。例如，如果这样执行程序：

```
py multiplicationTable.py 6
```

它应该创建一个图12-11所示的电子表格。

	A	B	C	D	E	F	G	H
1		**1**	**2**	**3**	**4**	**5**	**6**	
2	**1**	1	2	3	4	5	6	
3	**2**	2	4	6	8	10	12	
4	**3**	3	6	9	12	15	18	
5	**4**	4	8	12	16	20	24	
6	**5**	5	10	15	20	25	30	
7	**6**	6	12	18	24	30	36	
8								
9								

图12-11 在电子表格中生成的乘法表

行1和列A应该用做标签，应该使用粗体。

12.13.2 空行插入程序

创建一个程序blankRowInserter.py，它接受两个整数和一个文件名字符串作为命令行参数。我们将第一个整数称为N，第二个整数称为M。程序应该从第N行开始，在电子表格中插入M个空行。例如，如果这样执行程序：

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

执行之前和之后的电子表格，应该如图12-12所示。

A1 Potatoes

	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pe	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
4	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
5	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
6	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad
7	Asparagus	Fava bean	Green cab	Grapefruit	Ginger	Butterr
8	Avocados	Watermel	Eggplant	Grapes	Strawberri	Celery

A1 Potatoes

	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pe	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3						
4						
5	Fava bean	Spinach	Grapefruit	Grapes	Apricots	Papaya
6	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
7	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
8	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad

图12-12 之前（左边）和之后（右边）在第三行插入两个空行

程序可以这样写：读入电子表格的内容，然后在写入新的电子表格时，利用for循环拷贝前面N行。对于剩下的行，行号加上M，写入输出的电子表格。

12.13.3 电子表格单元格翻转程序

编写一个程序，翻转电子表格中行和列的单元格。例如，第5行第3列的值将出现在第3行第5列（反之亦然）。这应该针对电子表格中所有单元格进行。例如，之前和之后的电子表格应该看起来如图12-13所示。

A1 fx ITEM

	A	B	C	D	E	F	G	H	I	J
1	ITEM	SOLD								
2	Eggplant	334								
3	Cucumber	252								
4	Green cabl	238								
5	Eggplant	516								
6	Garlic	98								
7	Parsnips	16								
8	Asparagus	335								
9	Avocados	84								
10										

A1 fx ITEM

	A	B	C	D	E	F	G	H	I	J
1	ITEM	Eggplant	Cucumber	Green cabl	Eggplant	Garlic	Parsnips	Asparagus	Avocados	
2	SOLD	334	252	238	516	98	16	335	84	
3										
4										
5										
6										
7										
8										
9										
10										

图12-13 翻转之前（上面）和之后（下面）的电子表格

程序可以这样写：利用嵌套的for循环，将电子表格中的数据读入一个列表的列表。这个数据结构用sheetData[x][y]表示列x和行y处的单元格。然后，在写入新电子表格时，将sheetData[y][x]写入列x和行y处的单元格。

12.13.4 文本文件到电子表格

编写一个程序，读入几个文本文件的内容（可以自己创造这些文本文件），并将这些内容插入一个电子表格，每行写入一行文本。第一个文本文件中的行将写入列A中的单元格，第二个文本文件中的行将写入列B中的单元格，以此类推。

利用File对象的readlines()方法，返回一个字符串的列表，每个字符串就是文件中的一行。对于第一个文件，将第一行输出到列1行1。第二

行应该写入列1行2，以此类推。下一个用readlines()读入的文件将写入列2，再下一个写入列3，以此类推。

12.13.5 电子表格到文本文件

编写一个程序，执行前一个程序相反的任务。该程序应该打开一个电子表格，将列A中的单元格写入一个文本文件，将列B中的单元格写入另一个文本文件，以此类推。

第13章　处理PDF和Word文档

PDF和Word文档是二进制文件，所以它们比纯文本文件要复杂得多。除了文本之外，它们还保存了许多字体、颜色和布局信息。如果希望程序能读取或写入PDF和Word文档，需要做的就不只是将它们的文件名传递给open()。

好在，有一些Python模块。使得处理PDF和Word文档变得容易。本章将介绍两个这样的模块。

13.1 PDF文档

PDF表示Portable Document Format，使用.pdf文件扩展名。虽然PDF支持许多功能，但本章将专注于最常做的两件事：从PDF读取文本内容和从已有的文档生成新的PDF。

用于处理PDF的模块是PyPDF2。要安装它，就从命令行运行pip install PyPDF2。这个模块名称是区分大小写的，所以要确保y是小写，其他字母都是大写（请查看附录A，了解安装第三方模块的所有细节）。如果该模块安装正确，在交互式环境中运行import PyPDF2，应该不会显示任何错误。

13.1.1 从PDF提取文本

PyPDF2没有办法从PDF文档中提取图像、图表或其他媒体，但它可以提取文本，并将文本返回为Python字符串。为了开始学习PyPDF2的工作原理，我们将它用于一个示例PDF，如图13-1所示。

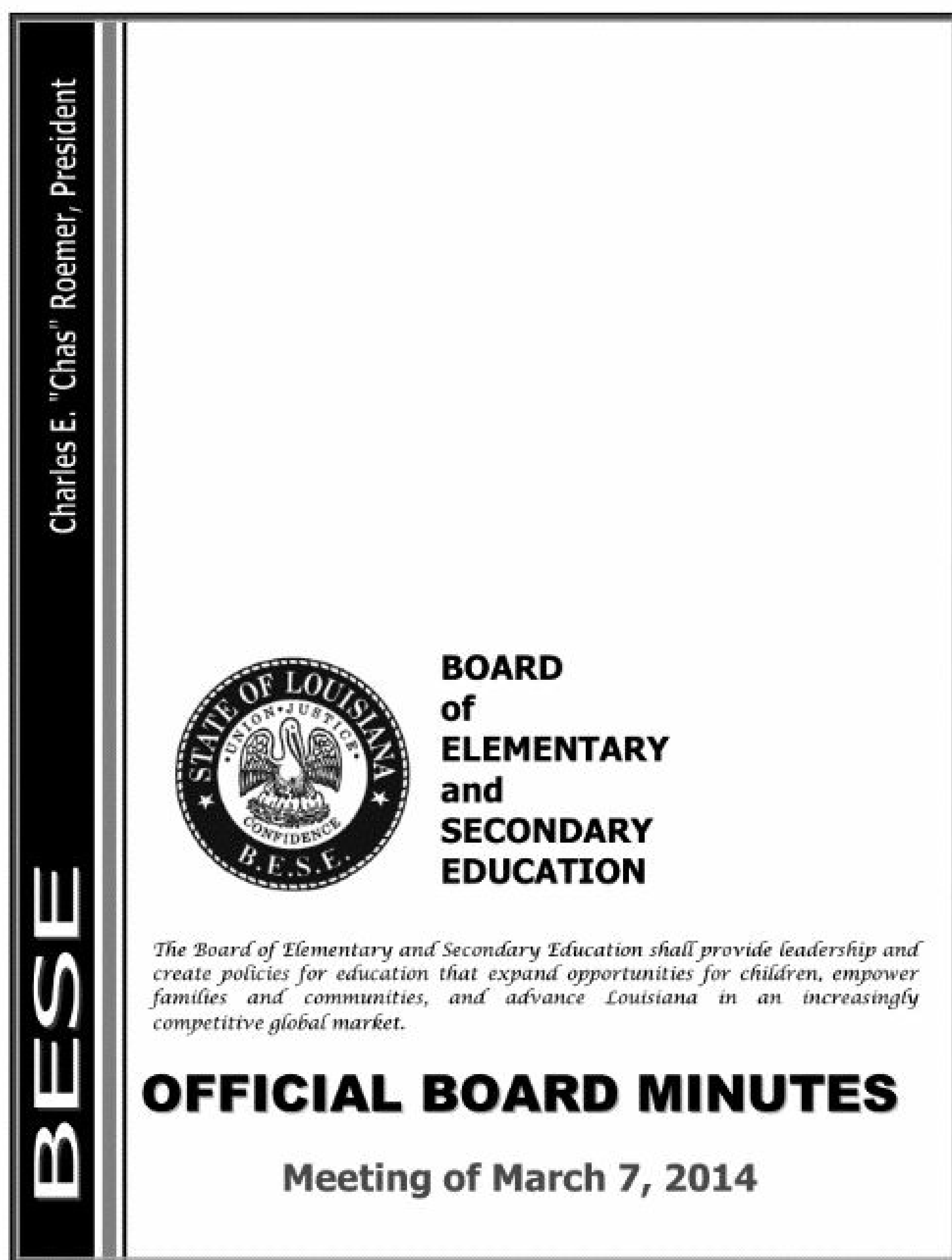


图13-1 PDF页面，我们将从中提取文本

有问题的PDF格式

虽然PDF文件对文本布局非常好，让人们很容易打印并阅读，但软件要将它们解析为纯文本却并不容易。因此，PyPDF2从PDF提取文本时可能会出错，甚至根本不能打开某些PDF。遗憾的是，你对此没有什么办法，PyPDF2可能就是不能处理某些PDF文件。话虽这样说，我至今没有发现不能用PyPDF2打开的PDF文件。

从http://nostarch.com/automatestuff/下载这个PDF文件，并在交互式环境中输入以下代码：

```
>>> import PyPDF2
  >>> pdfFileObj = open('meetingminutes.pdf', 'rb')
  >>> pdfReader = PyPDF2.PdfFileReader(pdfFileObj)
❶ >>> pdfReader.numPages
  19
❷ >>> pageObj = pdfReader.getPage(0)
❸ >>> pageObj.extractText()
  'OOFFFFIICCIIAALL  BBOOAARRDD   MMIINNUUTTEESS   Meeting of March 7, 2015
  \n      The Board of Elementary and Secondary Education shall provide lea
  and create policies for education that expand opportunities for children,
  empower families and communities, and advance Louisiana in an increasingl
  competitive global market. BOARD of ELEMENTARY and SECONDARY EDUCATION '
```

首先，导入PyPDF2模块。然后以读二进制模式打开meetingminutes.pdf，并将它保存在pdfFileObj中。为了取得表示这个PDF的PdfFileReader对象，调用PyPDF2. PdfFileReader()并向它传入pdfFileObj。将这个PdfFileReader对象保存在pdfReader中。

该文档的总页数保存在PdfFileReader对象的numPages属性中❶。示例PDF文档有19页，但我们只提取第一页的文本。

要从一页中提取文本，需要通过PdfFileReader对象取得一个Page对象，它表示PDF中的一页。可以调用PdfFileReader对象的getPage()方法❷，向它传入感兴趣的页码（在我们的例子中是0），从而取得Page对象。

PyPDF2在取得页面时使用从0开始的下标：第一页是0页，第二页是1页，以此类推。事情总是这样，即使文档中页面的页码不同。例如，假定你的PDF是从一个较长的报告中抽取出3页，它的页码分别是42、43和44，要取得这个文档的第一页，需要调用pdfReader.getPage(0)，而不是getPage(42)或getPage(1)。

在取得Page对象后，调用它的extractText()方法，返回该页文本的字符串❸。文本提取并不完美：该PDF中的文本Charles E.“Chas”Roemer, President，在函数返回的字符串中消失了，而且空格有时候也会没有。但是，这种近似的PDF文本内容，可能对你的程序来说已经足够了。

13.1.2 解密PDF

某些PDF文档有加密功能，以防止别人阅读，只有在打开文档时提供口令才能阅读。在交互式环境中输入以下代码，处理下载的PDF，它已经用口令rosebud加密：

```
>>> import PyPDF2
```

```
  >>> pdfReader = PyPDF2.PdfFileReader(open('encrypted.pdf', 'rb'))
❶ >>> pdfReader.isEncrypted
  True
  >>> pdfReader.getPage(0)
❷ Traceback (most recent call last):
    File "< pyshell#173>", line 1, in < module>
      pdfReader.getPage()
  --snip
--
  File "C:\Python34\lib\site-packages\PyPDF2\pdf.py", line 1173, in getObje
    raise utils.PdfReadError("file has not been decrypted")
  PyPDF2.utils.PdfReadError: file has not been decrypted
❸ >>> pdfReader.decrypt('rosebud')
  1
  >>> pageObj = pdfReader.getPage(0)
```

所有PdfFileReader对象都有一个isEncrypted属性，如果PDF是加密的，它就是True，如果不是，它就是False❶。在文件用正确的口令解密之前，尝试调用函数来读取文件，将会导致错误❷。

要读取加密的PDF，就调用decrypt()函数，传入口令字符串❸。在用正确的口令调用decrypt()后，你会看到调用getPage()不再导致错误。如果提供了错误的口令，decrypt()函数将返回0，并且getPage()会继续失败。请注意，decrypt()方法只解密了PdfFileReader对象，而不是实际的PDF文件。在程序中止后，硬盘上的文件仍然是加密的。程序下次运行时，仍然需要再次调用decrypt()。

13.1.3 创建PDF

在PyPDF2中，与PdfFileReader对象相对的是PdfFileWriter对象，它可以创建一个新的PDF文件。但PyPDF2不能将任意文本写入PDF，就像Python可以写入纯文本文件那样。PyPDF2写入PDF的能力，仅限于从其他PDF中拷贝页面、旋转页面、重叠页面和加密文件。

模块不允许直接编辑PDF。必须创建一个新的PDF，然后从已有的文档拷贝内容。本节的例子将遵循这种一般方式：

1．打开一个或多个已有的PDF（源PDF），得到PdfFileReader对象。

2．创建一个新的PdfFileWriter对象。

3．将页面从PdfFileReader对象拷贝到PdfFileWriter对象中。

4．最后，利用PdfFileWriter对象写入输出的PDF。

创建一个PdfFileWriter对象，只是在Python中创建了一个代表PDF文档的值，这并没有创建实际的PDF文件，要实际生成文件，必须调用PdfFileWriter对象的write()方法。

write()方法接受一个普通的File对象，它以写二进制的模式打开。你可以用两个参数调用Python的open()函数，得到这样的File对象：一个是要打开的PDF文件名字符串，一个是'wb'，表明文件应该以写二进制

的模式打开。

如果这听起来有些令人困惑，不用担心，在接下来的代码示例中，你会看到这种工作方式。

13.1.4 拷贝页面

可以利用PyPDF2，从一个PDF文档拷贝页面到另一个PDF文档。这让你能够组合多个PDF文件，去除不想要的页面，或调整页面的次序。

从http://nostarch.com/automatestuff/下载meetingminutes.pdf和meetingminutes2.pdf，放在当前工作目录中。在交互式环境中输入以下代码：

```
>>> import PyPDF2

>>> pdf1File = open('meetingminutes.pdf', 'rb')

>>> pdf2File = open('meetingminutes2.pdf', 'rb')

❶ >>> pdf1Reader = PyPDF2.PdfFileReader(pdf1File)

❷ >>> pdf2Reader = PyPDF2.PdfFileReader(pdf2File)
```

```
❸ >>> pdfWriter = PyPDF2.PdfFileWriter()

>>> for pageNum in range(pdf1Reader.numPages):

❹           pageObj = pdf1Reader.getPage(pageNum)

❺           pdfWriter.addPage(pageObj)

>>> for pageNum in range(pdf2Reader.numPages):

❹           pageObj = pdf2Reader.getPage(pageNum)
```

```
❺           pdfWriter.addPage(pageObj)

❻ >>> pdfOutputFile = open('combinedminutes.pdf', 'wb')

>>> pdfWriter.write(pdfOutputFile)

>>> pdfOutputFile.close()

>>> pdf1File.close()

>>> pdf2File.close()
```

以读二进制的模式打开两个PDF文件，将得到的两个File对象保存在pdf1File和pdf2File中。调用PyPDF2.PdfFileReader()，传入pdf1File，得到一个表示meetingminutes.pdf的PdfFileReader对象❶。再次调用PyPDF2.PdfFileReader()，传入pdf2File，得到一个表示meetingminutes2.pdf的PdfFileReader对象❷。然后创建一个新的PdfFileWriter对象，它表示一个空白的PDF文档❸。

接下来，从两个源PDF拷贝所有的页面，将它们添加到PdfFileWriter对象。在PdfFileReader对象上调用getPage()，取得Page对象❹。然后将这个Page对象传递给PdfFileWriter的addPage()方法❺。这些步骤先是针对pdf1Reader进行，然后再针对pdf2Reader进行。在拷贝页面完成后，向PdfFileWriter的write()方法传入一个File对象，写入一个新的PDF文档，名为combinedminutes.pdf❻。

注意

PyPDF2不能在PdfFileWriter对象中间插入页面，addPage()方法只能够在末尾添加页面。

现在你创建了一个新的PDF文件，将来自meetingminutes.pdf和meetingmin utes2.pdf的页面组合在一个文档中。要记住，传递给PyPDF2.PdfFileReader()的File对象，需要以读二进制的方式打开。即使用'rb'作为open()的第二个参数。类似的，传入PyPDF2.PdfFileWriter()的File对象需要以写二进制的模式打开，即使用'wb'。

13.1.5 旋转页面

利用rotateClockwise()和rotateCounterClockwise()方法，PDF文档的页面也可以旋转90度的整数倍。向这些方法传入整数90、180或270就可以了。在交互式环境中输入以下代码，同时将meetingminutes.pdf放在当前工作目录中：

```
>>> import PyPDF2

>>> minutesFile = open('meetingminutes.pdf', 'rb')
```

```
   >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)

❶ >>> page = pdfReader.getPage(0)

❷ >>> page.rotateClockwise(90)

   {'/Contents': [IndirectObject(961, 0), IndirectObject(962, 0),
   --snip

--
   }
   >>> pdfWriter = PyPDF2.PdfFileWriter()

   >>> pdfWriter.addPage(page)

❸>>> resultPdfFile = open('rotatedPage.pdf', 'wb')
```

```
>>> pdfWriter.write(resultPdfFile)

>>> resultPdfFile.close()

>>> minutesFile.close()
```

这里，我们使用getPage(0)来选择PDF的第一页❶，然后对该页调用rotateClockwise(90)❷。我们将旋转过的页面写入一个新的PDF文档，并保存为rotatedPage.pdf❸。

得到的PDF文件有一个页面，顺时针旋转了90度，如图13-2所示。rotateClockwise()和rotateCounterClockwise()的返回值包含许多信息，你可以忽略。

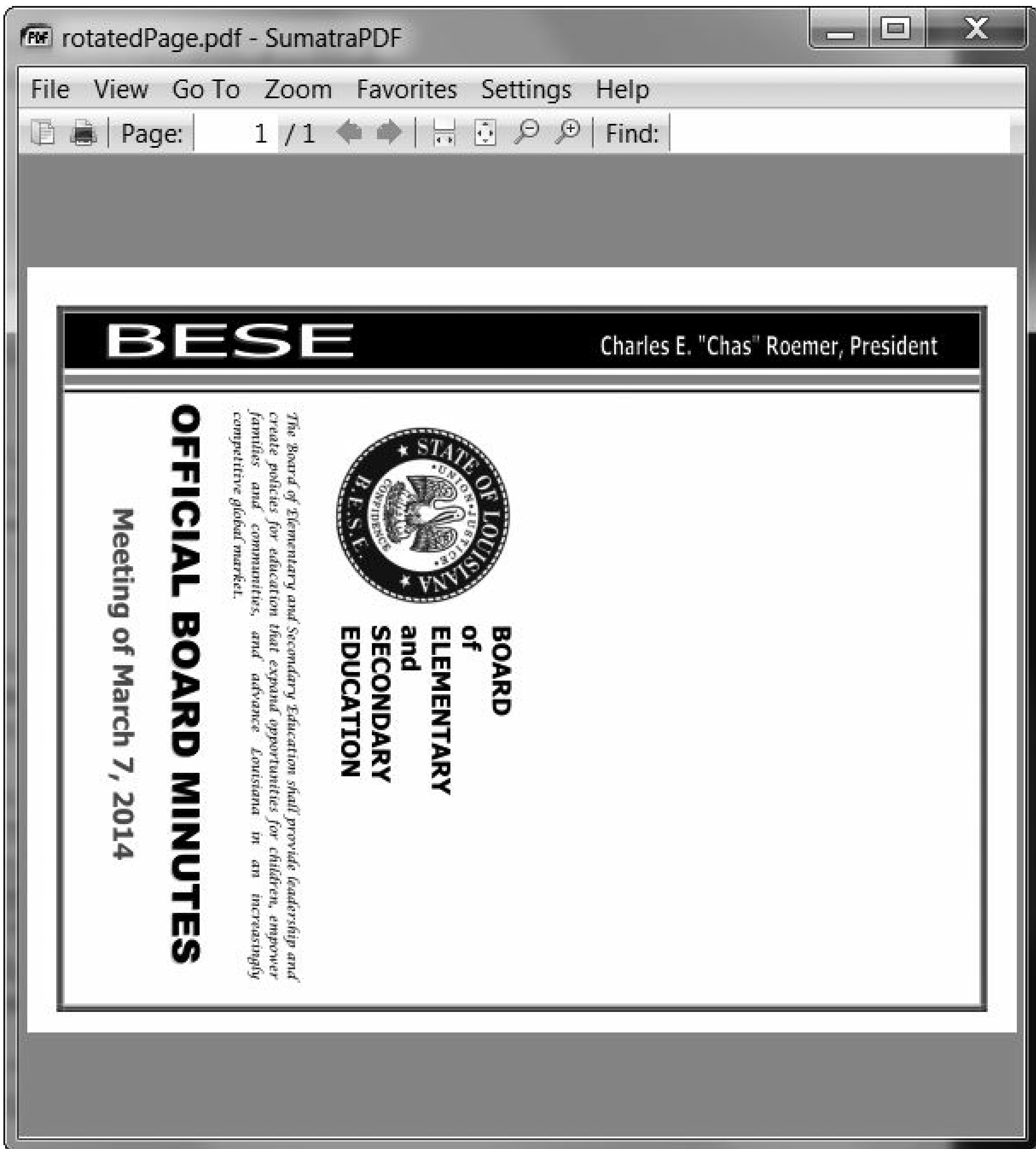


图13-2 rotatedPage.pdf文件，页面顺时针旋转了90度

13.1.6 叠加页面

PyPDF2也可以将一页的内容叠加到另一页上，这可以用来在页面上添加公司标志、时间戳或水印。利用Python，很容易为多个文件添加水印，并且只针对程序指定的页面添加。

从http://nostarch.com/automatestuff/下载watermark.pdf，将它和meetingminutes.pdf一起放在当前工作目录中。然后在交互式环境中输入以下代码：

```
  >>> import PyPDF2
  >>> minutesFile = open('meetingminutes.pdf', 'rb')
❶ >>> pdfReader = PyPDF2.PdfFileReader(minutesFile)
❷ >>> minutesFirstPage = pdfReader.getPage(0)
❸>>> pdfWatermarkReader = PyPDF2.PdfFileReader(open('watermark.pdf', 'rb'))
❹ >>> minutesFirstPage.mergePage(pdfWatermarkReader.getPage(0))
❺ >>> pdfWriter = PyPDF2.PdfFileWriter()
```

```
❻ >>> pdfWriter.addPage(minutesFirstPage)

❼ >>> for pageNum in range(1, pdfReader.numPages):

          pageObj = pdfReader.getPage(pageNum)

          pdfWriter.addPage(pageObj)

  >>> resultPdfFile = open('watermarkedCover.pdf', 'wb')

  >>> pdfWriter.write(resultPdfFile)

  >>> minutesFile.close()
```

```
>>> resultPdfFile.close()
```

这里我们生成了meetingminutes.pdf的PdfFileReader对象❶。调用getPage(0)，取得第一页的Page对象，并将它保存在minutesFirstPage中❷。然后生成了watermark.pdf的PdfFileReader对象❸，并在minutesFirstPage上调用mergePage()❹。传递给mergePage()的参数，是watermark.pdf第一页的Page对象。

既然我们已经在minutesFirstPage上调用了mergePage()，minutesFirstPage就代表加了水印的第一页。我们创建一个PdfFileWriter对象❺，并加入加了水印的第一页❻。然后循环遍历meetingminutes.pdf的剩余页面，将它们添加到PdfFileWriter对象中❼。最后，我们打开一个新的PDF文件watermarkedCover.pdf，并将PdfFileWriter的内容写入该文件。

图 13-3 展示了结果。新的 PDF 文件 watermarkedCover.pdf，包含meetingminutes.pdf的全部内容，并在第一页加了水印。



图13-3 最初的PDF（左边）、水印PDF（中间）以及合并的PDF（右边）

13.1.7 加密PDF

PdfFileWriter对象也可以为PDF文档进行加密。在交互式环境中输入以下代码：

```
>>> import PyPDF2

>>> pdfFile = open('meetingminutes.pdf', 'rb')

>>> pdfReader = PyPDF2.PdfFileReader(pdfFile)

>>> pdfWriter = PyPDF2.PdfFileWriter()

>>> for pageNum in range(pdfReader.numPages):

        pdfWriter.addPage(pdfReader.getPage(pageNum))
```

```
❶ >>> pdfWriter.encrypt('swordfish')

   >>> resultPdf = open('encryptedminutes.pdf', 'wb')

   >>> pdfWriter.write(resultPdf)

   >>> resultPdf.close()
```

在调用write()方法保存文件之前，调用encrypt()方法，传入口令字符串❶。PDF可以有一个用户口令（允许查看这个PDF）和一个拥有者口令（允许设置打印、注释、提取文本和其他功能的许可）。用户口令和拥有者口令分别是encrypt()的第一个和第二个参数。如果只传入一个字符串给encrypt()，它将作为两个口令。

在这个例子中，我们将meetingminutes.pdf的页面拷贝到PdfFileWriter对象。用口令swordfish加密了PdfFileWriter，打开了一个名为encryptedminutes.pdf的新PDF，将PdfFileWriter的内容写入新PDF。任何人要查看encryptedminutes.pdf，都必须输入这个口令。在确保文件的拷贝被正确加密后，你可能会删除原来的未加密的文件。

13.2 项目：从多个PDF中合并选择的页面

假定你有一个很无聊的任务，需要将几十个PDF文件合并成一个PDF文件。每一个文件都有一个封面作为第一页，但你不希望合并后的文件中重复出现这些封面。即使有许多免费的程序可以合并PDF，很多也只是简单的将文件合并在一起。让我们来写一个Python程序，定制需要合并到PDF中的页面。

总的来说，该程序需要完成：

- 找到当前工作目录中所有PDF文件。
- 按文件名排序，这样就能有序地添加这些PDF。
- 除了第一页之外，将每个PDF的所有页面写入输出的文件。

从实现的角度来看，代码需要完成下列任务：

- 调用os.listdir()，找到当前工作目录中的所有文件，去除掉非PDF文件。
- 调用Python的sort()列表方法，对文件名按字母排序。
- 为输出的PDF文件创建PdfFileWriter对象。
- 循环遍历每个PDF文件，为它创建PdfFileReader对象。
- 针对每个PDF文件，循环遍历每一页，第一页除外。
- 将页面添加到输出的PDF。
- 将输出的PDF写入一个文件，名为allminutes.pdf。

针对这个项目，打开一个新的文件编辑器窗口，将它保存为combinePdfs.py。

第**1**步：找到所有**PDF**文件

首先，程序需要取得当前工作目录中所有带.pdf扩展名的文件列表，并对它们排序。让你的代码看起来像这样：

```
  #! python3
  # combinePdfs.py - Combines all the PDFs in the current working directory
  # into a single PDF.

❶ import PyPDF2, os

  # Get all the PDF filenames.
  pdfFiles = []
  for filename in os.listdir('.'):
```

```
    if filename.endswith('.pdf'):
❷         pdfFiles.append(filename)
❸  pdfFiles.sort(key/str.lower)

❹ pdfWriter = PyPDF2.PdfFileWriter()

  # TODO: Loop through all the PDF files.

  # TODO: Loop through all the pages (except the first) and add them.

  # TODO: Save the resulting PDF to a file.
```

在#!行和介绍程序做什么的描述性注释之后，代码导入了os和PyPDF2模块❶。os.listdir('.')调用将返回当前工作目录中所有文件的列表。代码循环遍历这个列表，将带有.pdf扩展名的文件添加到pdfFiles中❷。然后，列表按照字典顺序排序，调用sort()时带有key/str.lower关键字参数❸。

代码创建了一个PdfFileWriter对象，保存合并后的PDF页面❹。最后，一些注释语句简要描述了剩下的程序。

第2步：打开每个PDF文件

现在，程序必须读取pdfFiles中的每个PDF文件。在程序中加入以下代码：

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory i
# a single PDF.

import PyPDF2, os

# Get all the PDFs filenames.
pdfFiles = []
--snip
```

```
--
# Loop through all the PDF files.

for filename in pdfFiles:

    pdfFileObj = open(filename, 'rb')

    pdfReader = PyPDF2.PdfFileReader(pdfFileObj)

    # TODO: Loop through all the pages (except the first) and add them.
# TODO: Save the resulting PDF to a file.
```

针对每个PDF文件，循环内的代码调用open()，以'wb'作为第二个参数，用读二进制的模式打开文件。open()调用返回一个 File 对象，它被传递给PyPDF2.PdfFileReader()，创建针对那个PDF文件的PdfFileReader对象。

第3步：添加每一页

针对每个PDF文件，需要循环遍历每一页，第一页除外。在程序中添加以下代码：

```
#! python3
```

```
# combinePdfs.py - Combines all the PDFs in the current working directory
# a single PDF.

import PyPDF2, os

--snip

--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip

--
    # Loop through all the pages (except the first) and add them.

❶     for pageNum in range(1, pdfReader.numPages):

          pageObj = pdfReader.getPage(pageNum)

          pdfWriter.addPage(pageObj)

# TODO: Save the resulting PDF to a file.
```

for循环内的代码将每个Page对象拷贝到PdfFileWriter对象。要记住，你需要跳过第一页。因为PyPDF2认为0是第一页，所以循环应该从1开始❶，然后向上增长到pdfReader.numPages中的整数，但不包括它。

第4步：保存结果

在这些嵌套的for循环完成后，pdfWriter变量将包含一个PdfFileWriter对象，合并了所有PDF的页面。最后一步是将这些内容写入硬盘上的一个文件。在程序中添加以下代码：

```
#! python3
# combinePdfs.py - Combines all the PDFs in the current working directory i
# a single PDF.
import PyPDF2, os

--snip





--

# Loop through all the PDF files.
for filename in pdfFiles:
--snip





--






# Loop through all the pages (except the first) and add them.
```

```
for pageNum in range(1, pdfReader.numPages):

--snip

--
# Save the resulting PDF to a file.

pdfOutput = open('allminutes.pdf', 'wb')

pdfWriter.write(pdfOutput)

pdfOutput.close()
```

向open()传入'wb'，以写二进制的模式打开输出PDF文件allminutes.pdf。然后，将得到的File对象传给write()方法，创建实际的PDF文件。调用close()方法，结束程序。

第5步：类似程序的想法

能够利用其他PDF文件的页面创建PDF文件，这让你的程序能完成以下任务：

- 从PDF文件中截取特定的页面。
- 重新调整PDF文件中页面的次序。
- 创建一个PDF文件，只包含那些具有特定文本的页面。文本由extractText()来确定。

13.3 Word文档

利用python-docx模块，Python可以创建和修改Word文档，它带有.docx文件扩展名。运行pip install python-docx，可以安装该模块（附录A介绍了安装第三方模块的细节）。

> **注意**
>
> OSI参考模型最初是在1983年由国际标准化组织出版，标准号为ISO 7498。在第一次用pip安装python-docx时，注意要安装python-docx，而不是docx。安装名称docx是指另一个模块，本书没有介绍。但是，在导入python-docx模块时，需要执行import docx，而不是import python-docx。

如果你没有Word软件，LibreOffice Writer和OpenOffice Writer都是免费的替代软件，它们可以在Windows、OS X和Linux上打开.docx文件。可以分别从https://www.libreoffice.org 和http://openoffice.org 下载它们。python-docx的完整文档在https://python-docx.readthedocs.org/ 。尽管有针对OS X平台的Word版本，但本章将使用Windows平台的Word。

和纯文本相比，.docx文件有很多结构。这些结构在python-docx中用3种不同的类型来表示。在最高一层，Document对象表示整个文档。Document对象包含一个Paragraph对象的列表，表示文档中的段落（用户在Word文档中输入时，如果按下回车，新的段落就开始了）。每个Paragraph对象都包含一个Run对象的列表。图13-4中的单句段落有4个Run对象。

A plain paragraph with some **bold** and some *italic*

Run Run Run Run

图13-4 一个Paragraph对象中识别的Run对象

Word文档中的文本不仅仅是字符串。它包含与之相关的字体、大小、颜色和其他样式信息。在Word中，样式是这些属性的集合。一个Run对象是相同样式文本的延续。当文本样式发生改变时，就需要一个新的Run对象。

13.3.1 读取Word文档

让我们尝试使用python-docx模块。从http://nostarch.com/automatestuff/下载demo.docx，并将它保存在当前工作目录中。然后在交互式环境中输入以下代码：

```
>>> import docx
❶ >>> doc = docx.Document('demo.docx')
❷ >>> len(doc.paragraphs)
7
❸ >>> doc.paragraphs[0].text
'Document Title'
❹ >>> doc.paragraphs[1].text
```

```
'A plain paragraph with some bold and some italic'
❺ >>> len(doc.paragraphs[1].runs)
4
❻ >>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
❼ >>> doc.paragraphs[1].runs[1].text
'bold'
❽ >>> doc.paragraphs[1].runs[2].text
' and some '
❾ >>> doc.paragraphs[1].runs[3].text
'italic'
```

在❶行，我们在Python中打开了一个.docx文件，调用docx.Document()，传入文件名demo.docx。这将返回一个Document对象，它有paragraphs属性，是Paragraph对象的列表。如果我们对doc.paragraphs调用len()，将返回7。这告诉我们，该文档有7个Paragraph

对象❷。每个Paragraph对象都有一个text属性，包含该段中文本的字符串（没有样式信息）。这里，第一个text属性包含'DocumentTitle'❸，第二个包含'A plain paragraph with some bold and some italic'❹。

每个Paragraph对象也有一个runs属性，它是Run对象的列表。Run对象也有一个text属性，包含那个延续中的文本。我们看看第二个Paragraph对象中的text属性，'A plain paragraph with some bold and some italic'。对这个Paragraph对象调用len()，结果告诉我们有4个Run对象❺。第一个对象包含'A plain paragraph with some '❻。然后，文本变为粗体样式，所以'bold'开始了一个新的Run对象❼。在这之后，文本又回到了非粗体的样式，这导致了第三个Run对象，' and some '❽。最后，第四个对象包含'italic'，是斜体样式❾。

有了python-docx，Python程序就能从.docx文件中读取文本，像其他的字符串值一样使用它。

13.3.2 从.docx文件中取得完整的文本

如果你只关心Word文档中的文本，不关心样式信息，就可以利用getText()函数。它接受一个.docx文件名，返回其中文本的字符串。打开一个新的文件编辑器窗口，输入以下代码，并保存为readDocx.py：

```
#! python3

import docx

def getText(filename):
    doc = docx.Document(filename)
    fullText = []
    for para in doc.paragraphs:
        fullText.append(para.text)
    return '\n'.join(fullText)
```

getText()函数打开了Word文档，循环遍历paragraphs列表中的所有Paragraph对象，然后将它们的文本添加到fullText列表中。循环结束后，fullText中的字符串连接在一起，中间以换行符分隔。

readDocx.py程序可以像其他模块一样导入。现在如果你只需要Word文档中的文本，就可以输入以下代码：

```
>>> import readDocx
>>> print(readDocx.getText('demo.docx'))
Document Title
A plain paragraph with some bold and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

也可以调整getText()，在返回字符串之前进行修改。例如，要让每一段缩进，就将文件中的append()调用替换为：

```
fullText.append(' ' + para.text)
```

要在段落之间增加空行，就将join()调用代码改成：

```
return '\n\n

'.join(fullText)
```

可以看到，只需要几行代码，就可以写出函数，读取.docx文件，根据需要返回它的内容字符串。

13.3.3 设置**Paragraph**和**Run**对象的样式

在Windows平台的Word中，你可以按下Ctrl-Alt-Shift-S，显示样式窗口并查看样式，如图13-5所示。在OS X上，可以点击ViewStyles菜单项，查看样式窗口。

Word和其他文字处理软件利用样式，保持类似类型的文本在视觉展现上一致，并易于修改。例如，也许你希望将内容段落设置为11点，Times New Roman，左对齐，右边不对齐的文本。可以用这些设置创建一种样式，将它赋给所有的文本段落。然后，如果稍后想改变文档中所有内容段落的展现形式，只要改变这种样式，所有段落都会自动更新。

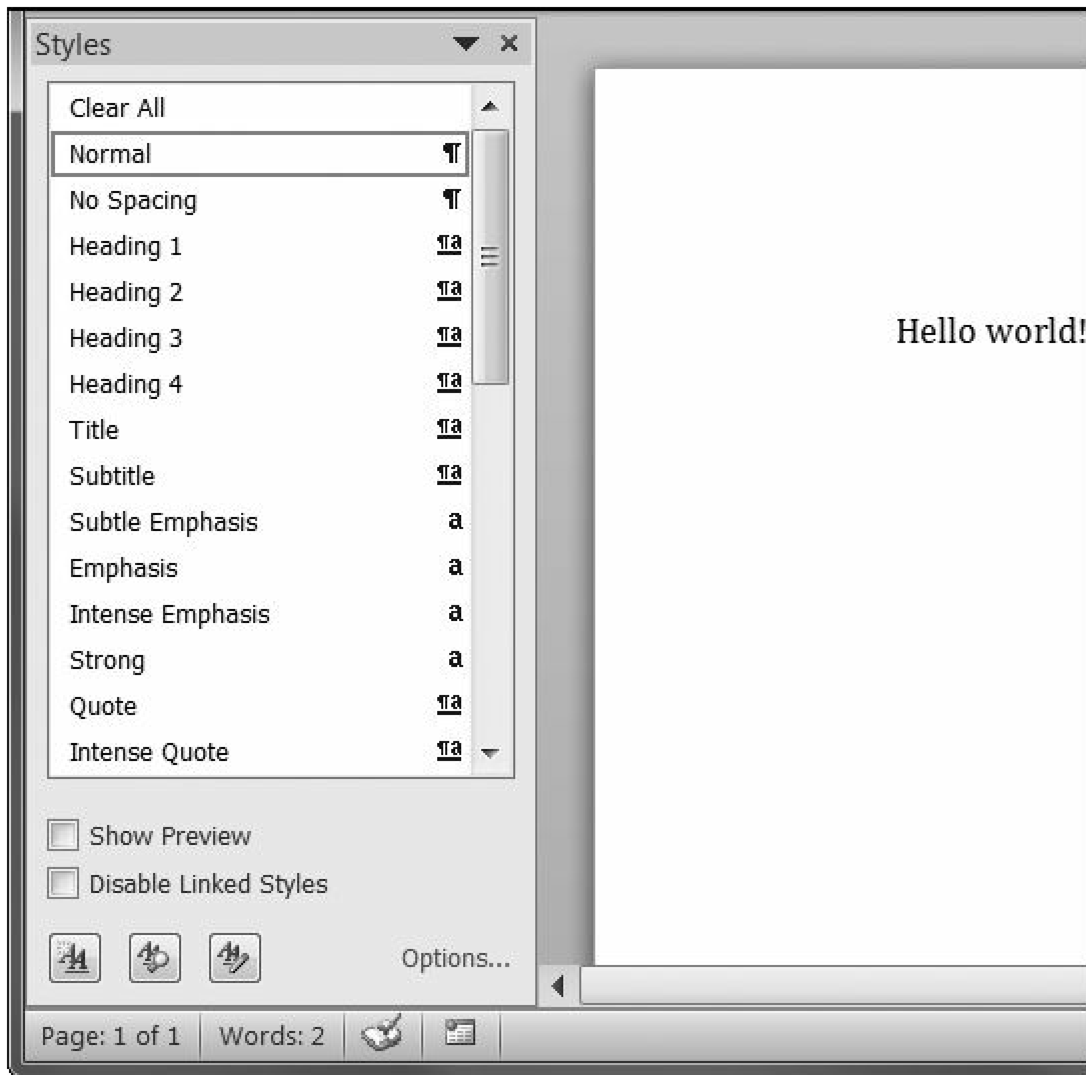


图13-5 在Windows平台上按下Ctrl-Alt-Shift-S，显示样式窗口

对于Word文档，有3种类型的样式：段落样式可以应用于Paragraph对象，字符样式可以应用于Run对象，链接的样式可以应用于这两种对象。可以将Paragraph和Run对象的style属性设置为一个字符串，从而设置样式。这个字符串应该是一种样式的名称。如果style被设置为None，就没有样式与Paragraph或Run对象关联。

默认Word样式的字符串如下：

```
'Normal'        'Heading5'        'ListBullet'        'ListParagraph'
'BodyText'      'Heading6'        'ListBullet2'       'MacroText'
'BodyText2'     'Heading7'        'ListBullet3'       'NoSpacing'
'BodyText3'     'Heading8'        'ListContinue'      'Quote'
'Caption'       'Heading9'        'ListContinue2'     'Subtitle'
'Heading1'      'IntenseQuote'    'ListContinue3'     'TOCHeading'
'Heading2'      'List'            'ListNumber'        'Title'
'Heading3'      'List2'           'ListNumber2'
'Heading4'      'List3'           'ListNumber3'
```

在设置style属性时，不要在样式名称中使用空格。例如，样式名称可能是Subtle Emphasis，你应该将属性设置为字符串'SubtleEmphasis'，而不是'Subtle Emphasis'。包含空格将导致Word误读样式名称，并且应用失败。

如果对Run对象应用链接的样式，需要在样式名称末尾加上'Char'。例如，对Paragraph对象设置Quote链接的样式，应该使用paragraphObj.style = 'Quote'。但对于Run对象，应该使用runObj.style = 'QuoteChar'。

在当前版本的python-docx (0.7.4)中，只能使用默认的Word样式，以及打开的文件中已有的样式，不能创建新的样式，但这一点在将来的模块版本中可能会改变。

13.3.4 创建带有非默认样式的**Word**文档

如果想要创建的Word文档使用默认样式以外的样式，就需要打开一个空白Word文档，通过点击样式窗口底部的New Style按钮，自己创建样式（图13-6展示了Windows平台上的情形）。

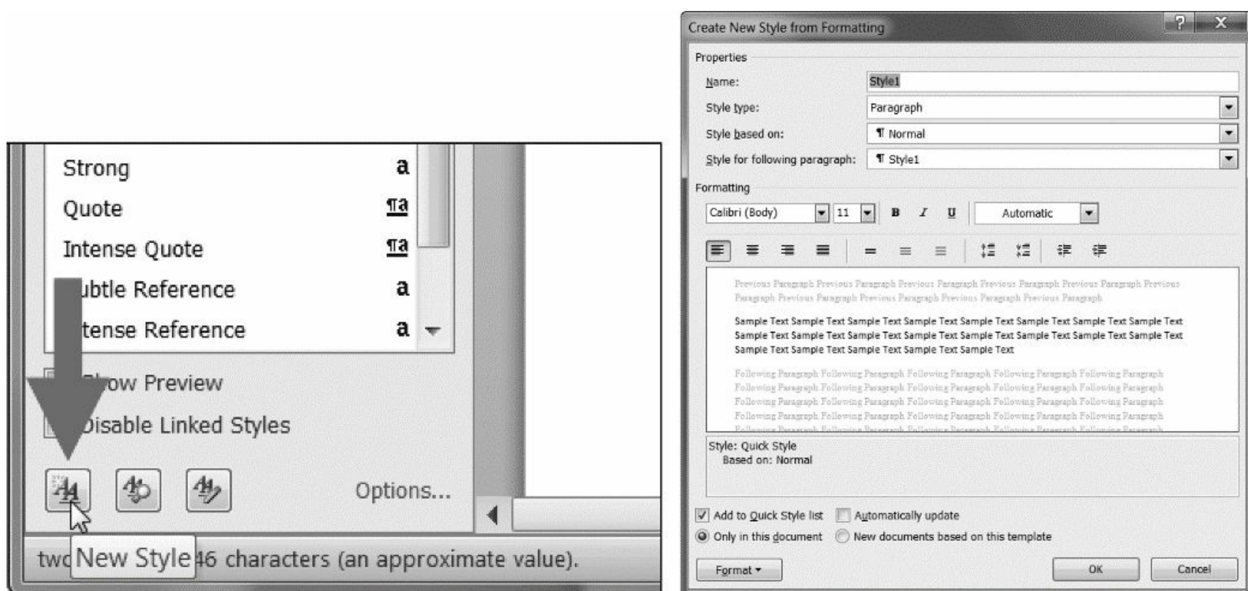


图13-6 新建样式按扭（左边）和“根据格式设置创建新样式”对话框（右边）

这将打开“Creat New Style from Formatting”对话框，在这里可以输入新样式。然后，回到交互式环境，用docx.Document()打开这个空白Word文档，利用它作为Word文档的基础。这种样式的名称现在就可以被python-docx使用了。

13.3.5 Run属性

通过text属性，Run可以进一步设置样式。每个属性都可以被设置为3个值之一：True（该属性总是启用，不论其他样式是否应用于该Run）、False（该属性总是禁用）或None（默认使用该Run被设置的任何属性）。

表13-1列出了可以在Run对象上设置的text属性。

表**13-1 Run**对象的**text**属性

属性	描述
bold	文本以粗体出现
italic	文本以斜体出现

underline	文本带下划线
strike	文本带删除线
double_strike	文本带双删除线
all_caps	文本以大写首字母出现
small_caps	文本以大写首字母出现，小写字母小两个点
shadow	文本带阴影
outline	文本以轮廓线出现，而不是实心
rtl	文本从右至左书写
imprint	文本以刻入页面的方式出现
emboss	文本以凸出页面的方式出现

例如，为了改变demo.docx的样式，在交互式环境中输入以下代码：

```
>>> doc = docx.Document('demo.docx')

>>> doc.paragraphs[0].text
```

```
'Document Title'
>>> doc.paragraphs[0].style
'Title'
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text, doc.
paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'QuoteChar'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
```

```
>>> doc.save('restyled.docx')
```

这里，我们使用了text和style属性，以便容易地看到文档的段落中有什么。我们可以看到，很容易将段落划分成Run，并单独访问每个Run。所以我们取得了第二段中的第一、第二和第四个Run，设置每个Run的样式，将结果保存到一个新文档。

文件顶部的单词Document Title将具有Normal样式，而不是Title样式。针对文本A plain paragraph的Run对象，将具有QuoteChar样式。针对单词bold和italic的两个Run对象，它们的underline属性设置为True。图13-7展示了文件中段落和Run的样式看起来的样子。

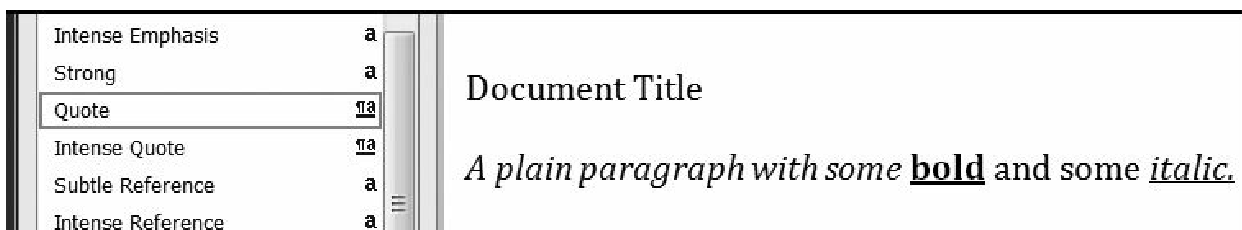


图13-7 restyled.docx文件

访问https://python-docx.readthedocs.org/en/latest/user/styles.html，你可以看到，python-docx使用样式的更完整文档。

13.3.6 写入Word文档

在交互式环境中输入以下代码：

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
< docx.text.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

要创建自己的.docx文件，就调用docx.Document()，返回一个新的、空白的Word Document对象。Document对象的add_paragraph()方法将一段新文本添加到文档中，并返回添加的Paragraph对象的引用。在添加完文本之后，向Document对象的save()方法传入一个文件名字符串，将Document对象保存到文件。

这将在当前工作目录中创建一个文件，名为helloworld.docx。如果打开它，就像图13-8的样子。

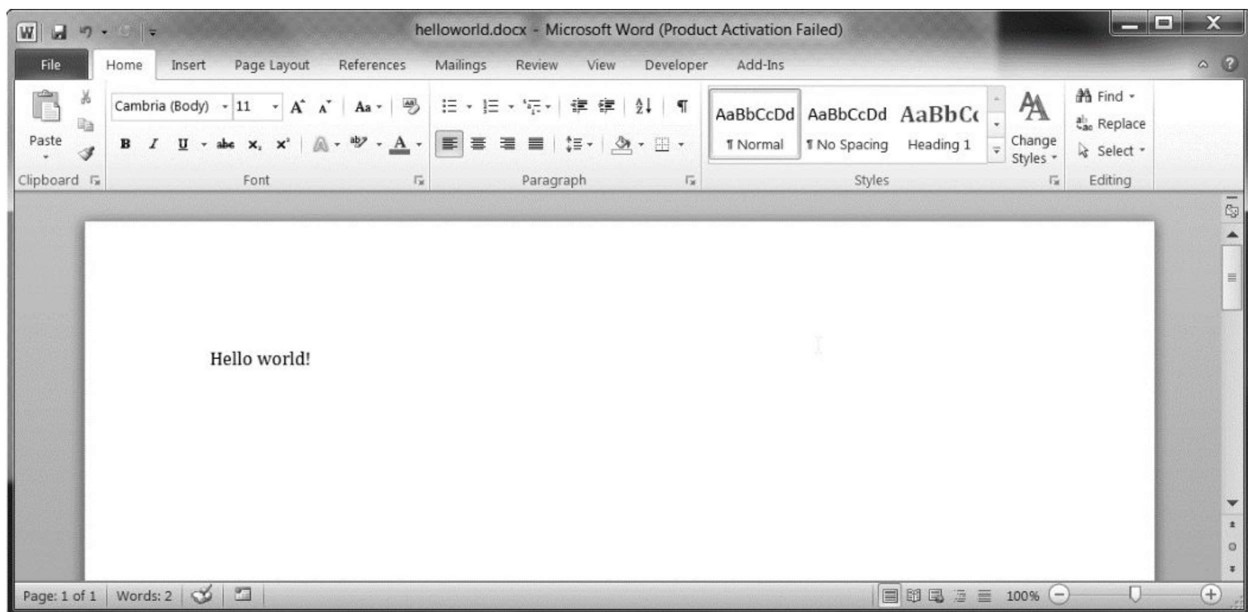


图13-8 利用add_paragraph('Hello world!')创建的Word文档

可以用新的段落文本，再次调用add_paragraph()方法，添加段落。或者，要在已有段落的末尾添加文本，可以调用Paragraph对象的add_run()方法，向它传入一个字符串。在交互式环境中输入以下代码：

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
< docx.text.Paragraph object at 0x000000000366AD30>
>>> paraObj1 = doc.add_paragraph('This is a second paragraph.')
```

```
>>> paraObj2 = doc.add_paragraph('This is a yet another paragraph.')
>>> paraObj1.add_run(' This text is being added to the second paragraph.')
< docx.text.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

得到的文本如图13-9所示。请注意，文本This text is being added to the second paragraph.被添加到paraObj1中的Paragraph对象中，它是添加到doc中的第二段。add_paragraph()和add_run()分别返回Paragraph和Run对象，这样你就不必多花一步来提取它们。

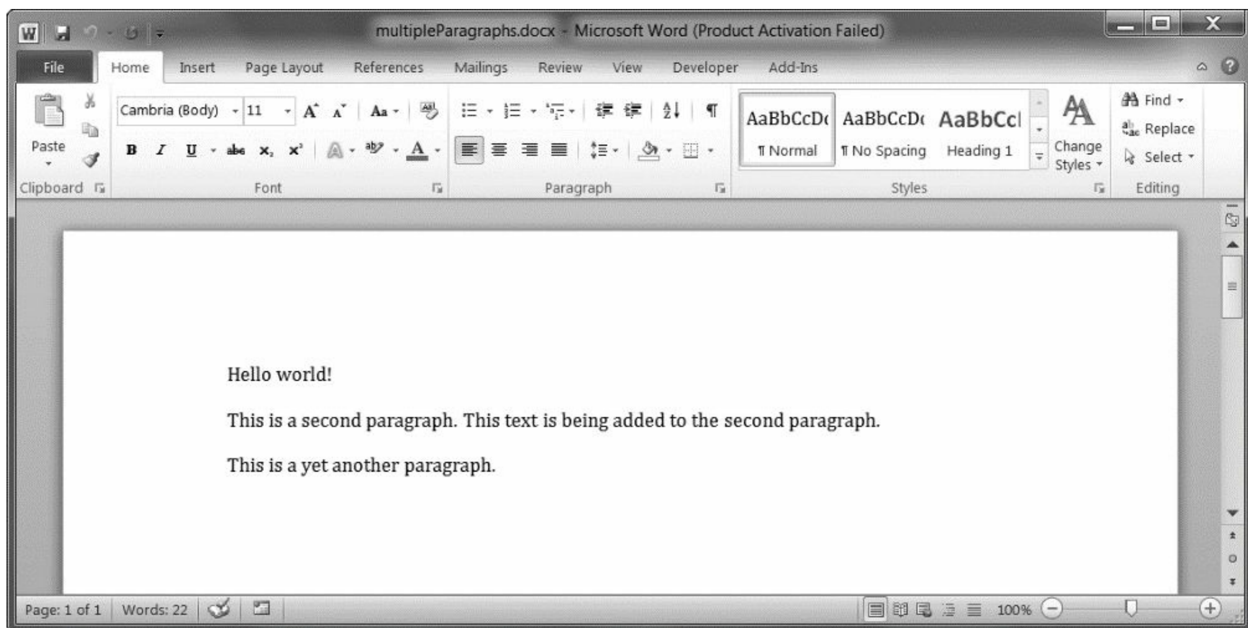


图13-9 添加了多个Paragraph和Run对象的文档

要记住，对于python-docx的0.5.3版本，新的Paragraph对象只能添加在文档的末尾，新的Run对象只能添加在Paragraph对象的末尾。

可以再次调用save()方法，保存所做的变更。

add_paragraph()和add_run()都接受可选的第二个参数，它是表示Paragraph或Run对象样式的字符串。例如：

```
>>> doc.add_paragraph('Hello world!', 'Title')
```

这一行添加了一段，文本是Hello world!，样式是Title。

13.3.7 添加标题

调用add_heading()将添加一个段落，并使用一种标题样式。在交互式环境中输入以下代码：

```
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
< docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
< docx.text.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
< docx.text.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
< docx.text.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
< docx.text.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

add_heading()的参数，是一个标题文本的字符串，以及一个从0到4的整数。整数0表示标题是Title样式，这用于文档的顶部。整数1到4是不同的标题层次，1是主要的标题，4是最低层的子标题。add_heading()返回一个Paragraph对象，让你不必多花一步从Document对象中提取它。

得到的headings.docx文件如图13-10所示。

Header 0

Header 1

Header 2

Header 3

Header 4

图13-10 带有标题0到4的headings.docx文档

13.3.8 添加换行符和换页符

要添加换行符（而不是开始一个新的段落），可以在Run对象上调用add_break()方法，换行符将出现在它后面。如果希望添加换页符，可以将docx.text.WD_BREAK.PAGE作为唯一的参数，传递给add_break()，就像下面代码中间所做的一样：

```
>>> doc = docx.Document()
```

```
>>> doc.add_paragraph('This is on the first page!')
< docx.text.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
< docx.text.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

这创建了一个两页的Word文档，第一页上是This is on the first page!，第二页上是This is on the second page!。虽然在文本This is on the first page!之后，第一页还有大量的空间，但是我们在第一段的第一个Run之后插入分页符，强制下一段落出现在新的页面中❶。

13.3.9 添加图像

Document对象有一个add_picture()方法，让你在文档末尾添加图像。假定当前工作目录中有一个文件zophie.png，你可以输入以下代码，在文档末尾添加zophie.png，宽度为1英寸，高度为4厘米（Word可

以同时使用英制和公制单位）：

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1),

height=docx.shared.Cm(4))

< docx.shape.InlineShape object at 0x00000000036C7D30>
```

第一个参数是一个字符串，表示图像的文件名。可选的width和height关键字参数，将设置该图像在文档中的宽度和高度。如果省略，宽度和高度将采用默认值，即该图像的正常尺寸。

你可能愿意用熟悉的单位来指定图像的高度和宽度，诸如英寸或厘米。所以在指定 width 和 height 关键字参数时，可以使用docx.shared.Inches()和docx.shared.Cm()函数。

13.4 小结

文本信息不仅仅是纯文本文件，实际上，很有可能更经常遇到的是PDF和Word文档。可以利用PyPDF2模块来读写PDF文档。遗憾的是，从PDF文档读取文本并非总是能得到完美转换的字符串，因为PDF文档的格式很复杂，某些PDF可能根本读不出来。在这种情况下，你就不太走运了，除非将来PyPDF2更新，支持更多的PDF功能。

Word文档更可靠，可以用python-docx模块来读取。可以通过Paragraph和Run对象来操作Word文档中的文本。可以设置这些对象的样式，尽管必须使用默认的样式，或文档中已有的样式。可以添加新的段落、标题、换行换页符和图像，尽管只能在文档的末尾。

在处理PDF和Word文档时有很多限制，这是因为这些格式的本意是很好地展示给人看，而不是让软件易于解析。下一章将探讨存储信息的另外两种常见格式：JSON和CSV文件。这些格式是设计给计算机使用的。你会看到，Python处理这些格式要容易得多。

13.5 习题

1．不能将PDF文件名的字符串传递给PyPDF2.PdfFileReader()函数。应该向该函数传递什么？

2．PdfFileReader()和PdfFileWriter()需要的File对象，应该以何种模式打开？

3．如何从PdfFileReader对象中取得第5页的Page对象？

4．什么PdfFileReader变量保存了PDF文档的页数？

5．如果PdfFileReader对象表示的PDF文档是用口令swordfish加密的，应该先做什么，才能从中取得Page对象？

6．使用什么方法来旋转页面？

7．什么方法返回文件demo.docx的Document对象？

8．Paragraph对象和Run对象之间的区别是什么？

9．doc变量保存了一个Document对象，如何从中得到Paragraph对象的列表？

10．哪种类型的对象具有bold、underline、italic、strike和outline变量？

11．bold变量设置为True、False或None，有什么区别？

12．如何为一个新Word文档创建Document对象？

13．doc变量保存了一个Document对象，如何添加一个文本是'Hello there!'的段落？

14．哪些整数表示Word文档中可用的标题级别？

13.6 实践项目

作为实践，编程完成下列任务。

13.6.1 PDF偏执狂

利用第9章的os.walk()函数编写一个脚本，遍历文件夹中的所有PDF（包含子文件夹），用命令行提供的口令对这些PDF加密。用原来的文件名加上_encrypted.pdf后缀，保存每个加密的PDF。在删除原来的文件之前，尝试用一个程序读取并解密该文件，确保它被正确的加密。

然后编写一个程序，找到文件夹中所有加密的PDF文件（包括它的子文件夹），利用提供的口令，创建PDF的解密拷贝。如果口令不对，程序应该打印一条消息，并继续处理下一个PDF文件。

13.6.2 定制邀请函，保存为Word文档

假设你有一个客人名单的文本文件。这个guests.txt文件每行有一个名字，像下面这样：

```
Prof. Plum
Miss Scarlet
Col. Mustard
Al Sweigart
RoboCop
```

写一个程序，生成定制邀请函的Word文档，如图13-11所示。

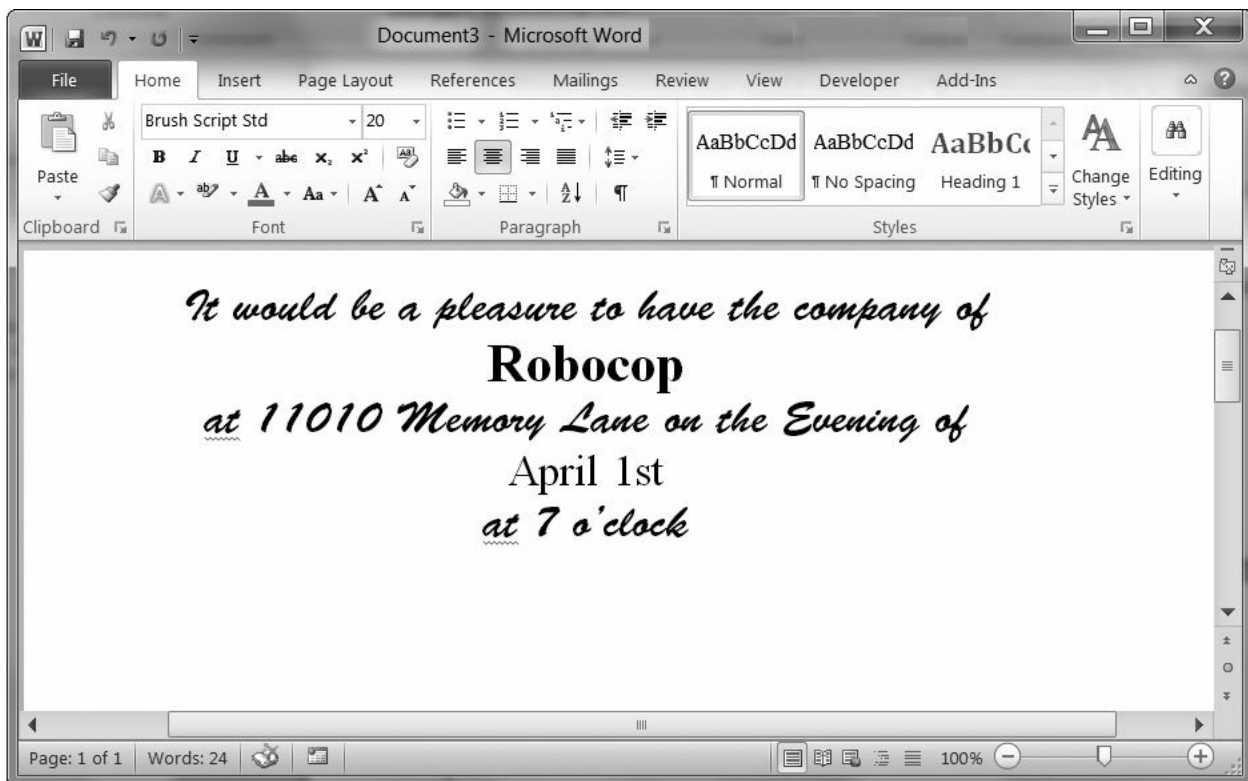


图13-11 定制的邀请函脚本生成的Word文档

因为python-docx只能使用Word文档中已经存在的样式，所以你必须先将这些样式添加到一个空白Word文件中，然后用python-docx打开该文件。在生成的Word文档中，每份邀请函应该占据一页，所以在每份邀请函的最后一段调用add_break()，添加分页符。这样，你只需要打开一份Word文档，就能打印所有的邀请函。

你可以从http://nostarch.com/automatestuff/下载示例guests.txt文件。

13.6.3 暴力PDF口令破解程序

假定有一个加密的PDF文件，你忘记了口令，但记得它是一个英语单词。尝试猜测遗忘的口令是很无聊的任务。作为替代，你可以写一个程序，尝试用所有可能的英语单词来解密这个PDF文件，直到找到有效的口令。这称为暴力口令攻击。从http://nostarch.com/automatestuff/下载文本文件dictionary.txt。这个字典文件包含44000多个英语单词，每个单词占一行。

利用第8章学过的文件读取技巧来读取这个文件，创建一个单词字

符串的列表。然后循环遍历这个列表中的每个单词，将它传递给decrypt()方法，如果这个方法返回整数 0，口令就是错的，程序应该继续尝试下一个口令。如果decrypt()返回1，程序就应该终止循环，打印出破解的口令。你应该尝试每个单词的大小写形式（在我的笔记本上，遍历来自字典文件的所有88000个大小写单词，只要几分钟时间。这就是不应该使用简单英语单词作为口令的原因）。

第14章　处理CSV文件和JSON数据

在第13章中，你学习了如何从PDF和Word文档中提取文本。这些文件是二进制格式，需要特殊的Python模块来访问它们的数据。CSV和JSON文件则不同，它们是纯文本文件。可以用文本编辑器察看它们，诸如IDLE的文件编辑器。但Python也有专门的csv和json模块，每个模块都提供了一些函数，帮助你处理这些文件格式。

CSV表示“Comma-Separated Values（逗号分隔的值）”，CSV文件是简化的电子表格，保存为纯文本文件。Python的csv模块让解析CSV文件变得容易。

JSON（发音为“JAY-sawn”或“Jason”，但如何发音并不重要。因为无论如何发音，都会有人说你发音错误）是一种格式，它以JavaScript源代码的形式，将信息保存在纯文本文件中。

JSON是JavaScript Object Notation的缩写不需要知道JavaScript编程语言，就可以使用JSON文件，但了解JSON格式是有用的，因为它用于许多Web应用程序中。

14.1 csv模块

CSV文件中的每行代表电子表格中的一行，逗号分割了该行中的单元格。例如，来自http://nostarch.com/automatestuff/的电子表格example.xlsx，在一个CSV文件中，看起来像这样：

```
4/5/2015 13:34,Apples,73
4/5/2015 3:41,Cherries,85
4/6/2015 12:46,Pears,14
4/8/2015 8:59,Oranges,52
4/10/2015 2:07,Apples,152
4/10/2015 18:10,Bananas,23
4/10/2015 2:40,Strawberries,98
```

我将使用这个文件作为本章的交互式环境的例子。可以从http://nostarch.com/ automatestuff/下载example.csv，或在文本编辑器中输入文本，并保存为example.csv。

CSV文件是简单的，缺少Excel电子表格的许多功能。例如，CSV文件中：

- 值没有类型，所有东西都是字符串；
- 没有字体大小或颜色的设置；
- 没有多个工作表；
- 不能指定单元格的宽度和高度；
- 不能合并单元格；
- 不能嵌入图像或图表。

CSV的文件的优势是简单。CSV文件被许多种类的程序广泛地支持，可以在文本编辑器中查看（包括IDLE的文件编辑器），它是表示电子表格数据的直接方式。CSV格式和它声称的完全一致：它就是一个文本文件，具有逗号分隔的值。

因为CSV文件就是文本文件，所以你可能会尝试将它们读入一个字符串，然后用第8章中学到的技术处理这个字符串。例如，因为CSV文件中的每个单元格有逗号分割，也许你可以只是对每行文本调用split()方法，来取得这些值。但并非CSV文件中的每个逗号，都表示两个单元格之间的分界。CSV文件也有自己的转义字符，允许逗号和其他字符作为值的一部分。split()方法不能处理这些转义字符。因为这些潜在的缺陷，所以总是应该使用csv模块来读写CSV文件。

14.1.1 Reader对象

要用csv模块从CSV文件中读取数据，需要创建一个Reader对象。Reader对象让你迭代遍历CSV文件中的每一行。在交互式环境中输入以下代码，同时将example.csv放在当前工作目录中：

```
❶ >>> import csv
```

```
❷ >>> exampleFile = open('example.csv')

❸ >>> exampleReader = csv.reader(exampleFile)

❹ >>> exampleData = list(exampleReader)

❺ >>> exampleData

  [['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'],
  ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'],
  ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23']
  ['4/10/2015 2:40', 'Strawberries', '98']]
```

csv模块是Python自带的，所以不需要安装就可以导入它❶。

要用csv模块读取CSV文件，首先用open()函数打开它❷，就像打开任何其他文本文件一样。但是，不用在open()返回的File对象上调用read()或readlines()方法，而是将它传递给csv.reader()函数❸。这将返回一个Reader对象，供你使用。请注意，不能直接将文件名字符串传递给csv.reader()函数。

要访问Reader对象中的值，最直接的方法，就是将它转换成一个普通Python列表，即将它传递给list()❹。在这个Reader对象上应用list()函数，将返回一个列表的列表。可以将它保存在变量exampleData中。在交互式环境中输入exampleData，将显示列表的列表❺。

既然已经将CSV文件表示为列表的列表，就可以用表达式exampleData [row][col]来访问特定行和列的值。其中，row是exampleData中一个列表的下标，col是该列表中你想访问的项的下标。在交互式环境中输入以下代码：

```
>>> exampleData[0][0]
'4/5/2015 13:34'
>>> exampleData[0][1]
'Apples'
>>> exampleData[0][2]
'73'
>>> exampleData[1][1]
'Cherries'
>>> exampleData[6][1]
'Strawberries'
```

exampleData[0][0]进入第一个列表，并给出第一个字符串。exampleData[0][2]进入第一个列表，并给出第三个字符串，以此类推。

14.1.2 在**for**循环中，从**Reader**对象读取数据

对于大型的CSV文件，你需要在一个for循环中使用Reader对象。这样避免将整个文件一次性装入内存。例如，在交互式环境中输入以下代码：

```
>>> import csv
>>> exampleFile = open('example.csv')
>>> exampleReader = csv.reader(exampleFile)
>>> for row in exampleReader:
        print('Row #' + str(exampleReader.line_num) + ' ' + str(row))
```

```
Row #1 ['4/5/2015 13:34', 'Apples', '73']
Row #2 ['4/5/2015 3:41', 'Cherries', '85']
Row #3 ['4/6/2015 12:46', 'Pears', '14']
Row #4 ['4/8/2015 8:59', 'Oranges', '52']
Row #5 ['4/10/2015 2:07', 'Apples', '152']
Row #6 ['4/10/2015 18:10', 'Bananas', '23']
Row #7 ['4/10/2015 2:40', 'Strawberries', '98']
```

在导入csv模块，并从CSV文件得到Reader对象之后，可以循环遍历Reader对象中的行。每一行是一个值的列表，每个值表示一个单元格。

print()函数将打印出当前行的编号以及该行的内容。要取得行号，就使用Reader对象的line_num变量，它包含了当前行的编号。

Reader对象只能循环遍历一次。要再次读取CSV文件，必须调用csv.reader，创建一个对象。

14.1.3 Writer对象

Writer 对象让你将数据写入 CSV 文件。要创建一个 Writer 对象，就使用csv.writer()函数。在交互式环境中输入以下代码。

```
  >>> import csv
❶ >>> outputFile = open('output.csv', 'w', newline='')
❷ >>> outputWriter = csv.writer(outputFile)
```

```
>>> outputWriter.writerow(['spam', 'eggs', 'bacon', 'ham'])
21
>>> outputWriter.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
32
>>> outputWriter.writerow([1, 2, 3.141592, 4])
16
>>> outputFile.close()
```

首先，调用open()并传入'w'，以写模式打开一个文件❶。这将创建对象。然后将它传递给csv.writer()❷，创建一个Writer对象。

在Windows上，需要为open()函数的newline关键字参数传入一个空字符串。这样做的技术原因超出了本书的范围。如果忘记设置newline关键字参数，output.csv中的行距将有两倍，如图14-1所示。

	A	B	C	D	E	F	G
1	42	2	3	4	5	6	7
2							
3	2	4	6	8	10	12	14
4							
5	3	6	9	12	15	18	21
6							
7	4	8	12	16	20	24	28
8							
9	5	10	15	20	25	30	35
10							

图14-1 如果你在open()中忘记了newline=''关键字参数，CSV文件将有两倍行距

Writer对象的writerow()方法接受一个列表参数。列表中的每个词，放在输出的CSV文件中的一个单元格中。writerow()函数的返回值，是写入文件中这一行的字符数（包括换行字符）。

这段代码生成的文件像下面这样：

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

请注意，Writer对象自动转义了'Hello, world!'中的逗号，在CSV文件中使用了双引号。模块csv让你不必自己处理这些特殊情况。

14.1.4 delimiter和lineterminator关键字参数

假定你希望用制表符代替逗号来分隔单元格，并希望有两倍行距。可以在交互式环境中输入下面这样的代码：

```
>>> import csv
```

```
>>> csvFile = open('example.tsv', 'w', newline='')
❶ >>> csvWriter = csv.writer(csvFile, delimiter='\t', lineterminator='\n\n'
>>> csvWriter.writerow(['apples', 'oranges', 'grapes'])
24
>>> csvWriter.writerow(['eggs', 'bacon', 'ham'])
17
>>> csvWriter.writerow(['spam', 'spam', 'spam', 'spam', 'spam', 'spam'])
32
>>> csvFile.close()
```

这改变了文件中的分隔符和行终止字符。分隔符是一行中单元格之间出现的字符。默认情况下，CSV文件的分隔符是逗号。行终止字符是出现在行末的字符。默认情况下，行终止字符是换行符。你可以利用csv.writer()的delimiter和lineterminator关键字参数，将这些字符改成不同的值。

传入delimeter='\t'和lineterminator='\n\n'❶，这将单元格之间的字符改变为制表符，将行之间的字符改变为两个换行符。然后我们调用writerow()三次，得到3行。

这产生了文件example.tsv，包含以下内容：

```
apples  oranges grapes

eggs    bacon   ham

spam    spam    spam    spam    spam    spam
```

既然单元格是由制表符分隔的，我们就使用文件扩展名.tsv，表示制表符分隔的值。

14.2 项目：从CSV文件中删除表头

假设你有一个枯燥的任务，要删除几百CSV文件的第一行。也许你会将它们送入一个自动化的过程，只需要数据，不需要每列顶部的表头。可以在Excel中打开每个文件，删除第一行，并重新保存该文件，但这需要几个小时。让我们写一个程序来做这件事。

该程序需要打开当前工作目录中所有扩展名为.csv的文件，读取CSV文件的内容，并除掉第一行的内容重新写入同名的文件。这将用新的、无表头的内容替换CSV文件的旧内容。

警告

> 与往常一样，当你写程序修改文件时，一定要先备份这些文件，以防万一你的程序没有按期望的方式工作。你不希望意外地删除原始文件。

总的来说，该程序必须做到以下几点：

- 找出当前工作目录中的所有CSV文件。
- 读取每个文件的全部内容。
- 跳过第一行，将内容写入一个新的CSV文件。

在代码层面上，这意味着该程序需要做到以下几点：

- 循环遍历从os.listdir()得到的文件列表，跳过非CSV文件。
- 创建一个CSV Reader对象，读取该文件的内容，利用line_num属性确定要跳过哪一行。
- 创建一个CSV Writer对象，将读入的数据写入新文件。

针对这个项目，打开一个新的文件编辑器窗口，并保存为removeCsvHeader.py。

第**1**步：循环遍历每个**CSV**文件

程序需要做的第一件事情，就是循环遍历当前工作目录中所有CSV文件名的列表。让removeCsvHeader.py看起来像这样：

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the curre
# working directory.

import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csvFilename in os.listdir('.'):
    if not csvFilename.endswith('.csv'):
❶        continue    # skip non-csv files

    print('Removing header from ' + csvFilename + '...')

    # TODO: Read the CSV file in (skipping first row).

    # TODO: Write out the CSV file.
```

```

```

os.makedirs()调用将创建headerRemoved文件夹，所有的无表头的CSV文件将写入该文件夹。针对os.listdir('.')进行for循环完成了一部分任务，但这会遍历工作目录中的所有文件，所以需要在循环开始处添加一些代码，跳过扩展名不是.csv的文件。如果遇到非CSV文件，continue语句❶让循环转向下一个文件名。

为了让程序运行时有一些输出，打印出一条消息说明程序在处理哪个CSV文件。然后，添加一些TODO注释，说明程序的其余部分应该做什么。

第2步：读入CSV文件

该程序不会从原来的CSV文件删除第一行。但是，它会创建新的CSV文件副本，不包含第一行。因为副本的文件名与原来的文件名一样，所以副本会覆盖原来的文件。

该程序需要一种方法，来知道它的循环当前是否在处理第一行。为removeCsvHeader.py添加以下代码。

```
#! python3
# removeCsvHeader.py - Removes the header from all CSV files in the current
# working directory.

--snip





--
# Read the CSV file in (skipping first row).






csvRows = []

```

```
csvFileObj = open(csvFilename)

readerObj = csv.reader(csvFileObj)

for row in readerObj:

    if readerObj.line_num == 1:

        continue     # skip first row

    csvRows.append(row)

csvFileObj.close()


# TODO: Write out the CSV file.
```

Reader对象的line_num属性可以用来确定当前读入的是CSV文件的哪一行。另一个for循环会遍历CSV Reader对象返回所有行，除了第一行，所有行都会添加到csvRows。

在for循环遍历每一行时，代码检查reader.line_num是否设为1。如果是这样，它执行continue，转向下一行，不将它添加到csvRows中。对于之后的每一行，条件永远是False，该行将添加到csvRows中。

第**3**步：写入**CSV**文件，没有第一行

现在csvRows包含了除第一行的所有行，该列表需要写入headerRemoved文件夹中的一个CSV文件。将以下代码添加到removeCsvHeader.py：

```
  #! python3
  # removeCsvHeader.py - Removes the header from all CSV files in the curre
  # working directory.
  --snip

--

  # Loop through every file in the current working directory.
❶ for csvFilename in os.listdir('.'):
      if not csvFilename.endswith('.csv'):
          continue # skip non-CSV files

      --snip

--

      # Write out the CSV file.
```

```
    csvFileObj = open(os.path.join('headerRemoved', csvFilename), 'w',
                 newline='')
    csvWriter = csv.writer(csvFileObj)
    for row in csvRows:
        csvWriter.writerow(row)
    csvFileObj.close()
```

CSV Writer对象利用csvFilename（这也是我们在CSV Reader中使用的文件名），将列表写入headerRemoved中的一个CSV文件。这将覆盖

原来的文件。

创建Writer对象后，我们循环遍历存储在csvRows中的子列表，将每个子列表写入该文件。

这段代码执行后，外层for循环❶将循环到os.listdir('.')中的下一个文件名。循环结束时，程序就结束了。

为了测试你的程序，从http://nostarch.com/automatestuff/下载removeCsvHeader.zip，将它解压缩到一个文件夹。在该文件夹中运行removeCsvHeader.py程序。输出将是这样的：

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
--snip

--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

这个程序应该在每次从CSV文件中删除第一行时，打印一个文件名。

第4步：类似程序的想法

针对CSV文件写的程序类似于针对Excel文件写的程序，因为它们都是电子表格文件。你可以编程完成以下任务：

- 在一个CSV文件的不同行，或多个CSV文件之间比较数据。
- 从CSV文件拷贝特定的数据到Excel文件，或反过来。
- 检查CSV文件中无效的数据或格式错误，并向用户提醒这些错误。
- 从CSV文件读取数据，作为Python程序的输入。

14.3 JSON和API

JavaScript对象表示法是一种流行的方式，将数据格式化，成为人可读的字符串。JSON是JavaScript程序编写数据结构的原生方式，通常类似于Python的pprint()函数产生的结果。不需要了解JavaScript，也能处理JSON格式的数据。

下面是JSON格式数据的一个例子：

```
{"name": "Zophie", "isCat": true,
 "miceCaught": 0, "napsTaken": 37.5,
 "felineIQ": null}
```

了解JSON是很有用，因为很多网站都提供JSON格式的内容，作为程序与网站交互的方式。这就是所谓的提供"应用程序编程接口（API）"。访问API和通过URL访问任何其他网页是一样的。不同的是，API返回的数据是针对机器格式化的（例如用JSON），API不是人容易阅读的。

许多网站用JSON格式提供数据。Facebook、Twitter、Yahoo、Google、Tumblr、Wikipedia、Flickr、Data.gov、Reddit、IMDb、Rotten Tomatoes、LinkedIn和许多其他流行的网站，都提供API让程序使用。有些网站需要注册，这几乎都是免费的。你必须找到文档，了解程序需要请求什么 URL 才能获得想要的数据，以及返回的JSON数据结构的一般格式。这些文档应在提供API的网站上提供，如果它们有"开发者"页面，就去那里找找。

利用API，可以编程完成下列任务：

- 从网站抓取原始数据（访问API通常比下载网页并用Beautiful Soup解析HTML更方便）。
- 自动从一个社交网络账户下载新的帖子，并发布到另一个账户。例如，可以把tumblr的帖子上传到Facebook。
- 从IMDb、Rotten Tomatoes和维基百科提取数据，放到计算机的一个文本文件中，为你个人的电影收藏创建一个"电影百科全书"。

可以在http://nostarch.com/automatestuff/的资源中看到JSON API的一些例子。

14.4 json模块

Python的json模块处理了JSON数据字符串和Python值之间转换的所有细节，得到了json.loads()和json.dumps()函数。JSON不能存储每一种Python值，它只能包含以下数据类型的值：字符串、整型、浮点型、布尔型、列表、字典和NoneType。JSON不能表示Python特有的对象，如File对象、CSV Reader或Writer对象、Regex对象或Selenium WebElement对象。

14.4.1 用loads()函数读取JSON

要将包含JSON数据的字符串转换为Python的值，就将它传递给json.loads()函数（这个名字的意思是“load string”，而不是“loads”）。在交互式环境中输入以下代码：

```
>>> stringOfJsonData = '{"name": "Zophie", "isCat": true, "miceCaught": 0,
"felineIQ": null}'
>>> import json
>>> jsonDataAsPythonValue = json.loads(stringOfJsonData)
```

```
>>> jsonDataAsPythonValue

{'isCat': True, 'miceCaught': 0, 'name': 'Zophie', 'felineIQ': None}
```

导入json模块后，就可以调用loads()，向它传入一个JSON数据字符串。请注意，JSON字符串总是用双引号。它将该数据返回为一个Python字典。Python字典是没有顺序的，所以如果打印jsonDataAsPythonValue，键-值对可能以不同的顺序出现。

14.4.2 用**dumps**函数写出**JSON**

json.dumps()函数（它表示“dump string”，而不是“dumps”）将一个Python值转换成JSON格式的数据字符串。在交互式环境中输入以下代码：

```
>>> pythonValue = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie',

'felineIQ': None}

>>> import json

>>> stringOfJsonData = json.dumps(pythonValue)
```

```
>>> stringOfJsonData
'{"isCat": true, "felineIQ": null, "miceCaught": 0, "name": "Zophie" }'
```

该值只能是以下基本Python数据类型之一：字典、列表、整型、浮点型、字符串、布尔型或None。

14.5 项目：取得当前的天气数据

检查天气似乎相当简单：打开Web浏览器，点击地址栏，输入天气网站的URL（或搜索一个，然后点击链接），等待页面加载，跳过所有的广告等。

其实，如果有一个程序，下载今后几天的天气预报，并以纯文本打印出来，就可以跳过很多无聊的步骤。该程序利用第11章介绍的requests模块，从网站下载数据。

总的来说，该程序将执行以下操作：

- 从命令行读取请求的位置。
- 从OpenWeatherMap.org下载JSON天气数据。
- 将JSON数据字符串转换成Python的数据结构。
- 打印今天和未来两天的天气。

因此，代码需要完成以下任务：

- 连接sys.argv中的字符串，得到位置。
- 调用requests.get()，下载天气数据。
- 调用json.loads()，将JSON数据转换为Python数据结构。

- 打印天气预报。

针对这个项目，打开一个新的文件编辑器窗口，并保存为quickWeather.py。

第1步：从命令行参数获取位置

该程序的输入来自命令行。让quickWeather.py看起来像这样：

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line

import json, requests, sys

# Compute location from command line arguments.
if len(sys.argv) <  2:
    print('Usage: quickWeather.py location')
    sys.exit()
location = ' '.join(sys.argv[1:])

# TODO: Download the JSON data from OpenWeatherMap.org's API.

# TODO: Load JSON data into a Python variable.
```

在Python中，命令行参数存储在sys.argv列表里。#!行和import语句之后，程序会检查是否有多个命令行参数（回想一下，sys.argv中至少有一个元素sys.argv[0]，它包含了Python脚本的文件名）。如果该列表中只有一个元素，那么用户没有在命令行中提供位置，程序向用户提供“Usage（用法）”信息，然后结束。

命令行参数以空格分隔。命令行参数San Francisco, CA将使sys.argv中保存['quickWeather.py', 'San', 'Francisco,', 'CA']。因此，调用join()方法，将sys.argv中除第一个字符串以外的字符串连接起来。将连接的字符串存储在变量location中。

第2步：下载JSON数据

OpenWeatherMap.org提供了JSON格式的实时天气信息。你的程序只需要下载页面http://api.openweathermap.org/data/2.5/forecast/daily?q=< Location>&cnt=3，其中< Location>是想知道天气的城市。将以下代码添加到quickWeather.py中。

```
#! python3
# quickWeather.py - Prints the weather for a location from the command line

--snip

--

# Download the JSON data from OpenWeatherMap.org's API.

url ='http://api.openweathermap.org/data/2.5/forecast/daily?q=%s&cnt=3' % (

response = requests.get(url)

response.raise_for_status()

# TODO: Load JSON data into a Python variable.
```

我们从命令行参数中得到了 location。为了生成要访问的网址，我们利用%s占位符，将location中保存的字符串插入URL字符串的那个位置。结果保存在url中，并将url传入requests.get()。requests.get()调用返回一个Response对象，它可以通过调用raise_for_status()来检查错误。如果不发生异常，下载的文本将保存在response.text中。

第**3**步：加载**JSON**数据并打印天气

response.text成员变量保存了一个JSON格式数据的大字符串。要将它转换为Python值，就调用json.loads()函数。JSON数据会像这样：

```
{'city': {'coord': {'lat': 37.7771, 'lon': -122.42},
          'country': 'United States of America',
          'id': '5391959',
          'name': 'San Francisco',
          'population': 0},
 'cnt': 3,
 'cod': '200',
 'list': [{'clouds': 0,
           'deg': 233,
           'dt': 1402344000,
           'humidity': 58,
           'pressure': 1012.23,
           'speed': 1.96,
           'temp': {'day': 302.29,
                    'eve': 296.46,
                    'max': 302.29,
                    'min': 289.77,
                    'morn': 294.59,
                    'night': 289.77},
           'weather': [{'description': 'sky is clear',
                        'icon': '01d',
--snip



--
```

可以将weatherData传入pprint.pprint，查看这个数据。你可能要查

找http:// openweathermap.org/ ，找到关于这些字段含义的文档。例如，在线文档会告诉你，'day'后面的302.29是白天的开尔文温度，而不是摄氏或华氏温度。

你想要的天气描述在'main'和'description'之后。为了整齐地打印出来，在quickWeather.py中添加以下代码。

```
#! python3
# quickWeather.py - Prints the weather for a location from the command li

--snip

--

# Load JSON data into a Python variable.

weatherData = json.loads(response.text)

# Print weather descriptions.

❶ w = weatherData['list']

print('Current weather in %s:' % (location))
```

```
print(w[0]['weather'][0]['main'], '-', w[0]['weather'][0]['description'])

print()

print('Tomorrow:')

print(w[1]['weather'][0]['main'], '-', w[1]['weather'][0]['description'])

print()

print('Day after tomorrow:')

print(w[2]['weather'][0]['main'], '-', w[2]['weather'][0]['description'])
```

请注意，代码将weatherData['list']保存在变量w中，这将节省一些打字时间❶。可以用w[0]、w[1]和w[2]来取得今天、明天和后天天气的字典。这些字典都有'weather'键，其中包含一个列表值。你感兴趣的是第一个列表项（一个嵌套的字典，包含几个键），下标是0。这里，我们打印出保存在'main'和'description'键中的值，用连字符隔开。

如果用命令行参数quickWeather.py San Francisco, CA运行这个程序，输出看起来是这样的：

```
Current weather in San Francisco, CA:
Clear - sky is clear

Tomorrow:
Clouds - few clouds

Day after tomorrow:
Clear - sky is clear
```

（天气是我喜欢住在旧金山的原因之一！）

第4步：类似程序的想法

访问气象数据可以成为多种类型程序的基础。你可以创建类似程序，完成以下任务：

- 收集几个露营地点或远足路线的天气预报，看看哪一个天气最好。
- 如果需要将植物移到室内，安排一个程序定期检查天气并发送霜冻警报（第15章介绍了定时调度，第16章介绍了如何发送电子邮件）。
- 从多个站点获得气象数据，同时显示，或计算并显示多个天气预报的平均值。

14.6 小结

CSV和JSON是常见的纯文本格式，用于保存数据。它们很容易被

程序解析，同时仍然让人可读，所以它们经常被用作简单的电子表格或网络应用程序的数据。csv和json模块大大简化了读取和写入CSV和JSON文件的过程。

前面几章教你如何利用Python从各种各样的文件格式的解析信息。一个常见的任务是接受多种格式的数据，解析它，并获得需要的特定信息。这些任务往往非常特别，商业软件并不是最有帮助的。通过编写自己的脚本，可以让计算机处理大量以这些格式呈现的数据。

在第15章，你将从数据格式中挣脱，学习如何让程序与你通信，发送电子邮件和文本消息。

14.7 习题

1．哪些功能是Excel电子表格有，而CSV电子表格没有？

2．向csv.reader()和csv.writer()传入什么，来创建Reader和Writer对象？

3．对于Reader和Writer对象，File对象需要以什么模式打开？

4．什么方法接受一个列表参数，并将其写入CSV文件？

5．delimiter和lineterminator关键字参数有什么用？

6．什么函数接受一个JSON数据的字符串，并返回一个Python数据结构？

7．什么函数接受一个Python数据结构，并返回一个JSON数据的字符串？

14.8 实践项目

作为实践，编程完成下列任务。

Excel到CSV的转换程序

Excel可以将电子表格保存为CSV文件，只要点几下鼠标，但如果有几百个Excel文件要转换为CSV，就需要点击几小时。利用第12章的openpyxl模块，编程读取当前工作目录中的所有Excel文件，并输出为CSV文件。

一个Excel文件可能包含多个工作表，必须为每个表创建一个CSV文件。CSV文件的文件名应该是< Excel文件名>_< 表标题>.csv，其中< Excel文件名>是没有扩展名的Excel文件名（例如'spam_data'，而不是'spam_data.xlsx'），< 表标题>是Worksheet对象的title变量中的字符串。

该程序将包含许多嵌套的for循环。该程序的框架看起来像这样：

```
for excelFile in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheetName in wb.get_sheet_names():
        # Loop through every sheet in the workbook.
        sheet = wb.get_sheet_by_name(sheetName)

        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for rowNum in range(1, sheet.get_highest_row() + 1):
            rowData = [] # append each cell to this list
            # Loop through each cell in the row.
            for colNum in range(1, sheet.get_highest_column() + 1):
                # Append each cell's data to rowData.

            # Write the rowData list to the CSV file.

        csvFile.close()
```

从http://nostarch.com/automatestuff/下载ZIP文件excelSpreadsheets.zip，将这些电子表格解压缩到程序所在的目录中。可以使用这些文件来测试程序。

第15章　保持时间、计划任务和启动程序

坐在电脑前运行程序是不错的，但在你没有直接监督时运行程序，也是有用的。计算机的时钟可以调度程序，在特定的时间和日期运行，或定期运行。例如，程序可以每小时抓取一个网站，检查变更，或在凌晨4点你睡觉时，执行CPU密集型任务。Python的time和datetime模块提供了这些函数。

利用subprocess和threading模块，你也可以编程按时启动其他程序。通常，编程最快的方法是利用其他人已经写好的应用程序。

15.1 time模块

计算机的系统时钟设置为特定的日期、时间和时区。内置的time模块让Python程序能读取系统时钟的当前时间。在time模块中，time.time()和time.sleep()函数是最有用的模块。

15.1.1 time.time()函数

Unix纪元是编程中经常参考的时间：1970年1月1日0点，即协调世界时（UTC）。time.time()函数返回自那一刻以来的秒数，是一个浮点值（回想一下，浮点值只是一个带小数点的数）。这个数字称为UNIX纪元时间戳。例如，在交互式环境中输入以下代码：

```
>>> import time
>>> time.time()
```

```
1425063955.068649
```

这里，我在2015年2月27日，太平洋标准时间11:05（或7:05 PM UTC），调用time.time()。返回值是Unix纪元的那一刻与time.time()被调用的那一刻之间的秒数。

注意

交互式环境的例子得到的日期和时间，是我在2015年2月写这一章的时间。除非你是时间旅行者，否则得到的日期和时间会不同。

纪元时间戳可以用于剖析代码，也就是测量一段代码的运行时间。如果在代码块开始时调用time.time()，并在结束时再次调用，就可以用第二个时间戳减去第一个，得到这两次调用之间经过的时间。例如，打开一个新的文件编辑器窗口，然后输入以下程序：

```
  import time
❶ def calcProd():
      # Calculate the product of the first 100,000 numbers.
      product = 1
      for i in range(1, 100000):
          product = product * i
      return product

❷ startTime = time.time()
  prod = calcProd()
❸ endTime = time.time()
❹ print('The result is %s digits long.' % (len(str(prod))))
❺ print('Took %s seconds to calculate.' % (endTime - startTime))
```

在❶行，我们定义了函数calcProd()，循环遍历1至99999的整数，返回它们的乘积。在❷行，我们调用time.time()，将结果保存在startTime中。调用calcProd()后，我们再次调用time.time()，将结果保存endTime中

❸。最后我们打印calcProd()返回的乘积的长度❹，以及运行calcProd()的时间❺。

将该程序保存为calcProd.py，并运行它。输出看起来像这样：

```
The result is 456569 digits long.
Took 2.844162940979004 seconds to calculate.
```

注意

另一种剖析代码的方法是利用cProfile.run()函数。与简单的time.time()技术相比，它提供了详细的信息。cProfile.run()函数在https://docs.python.org/3/library/ profile.html 有解释。

15.1.2 time.sleep()函数

如果需要让程序暂停一下，就调用time.sleep()函数，并传入希望程序暂停的秒数。在交互式环境中输入以下代码：

```
  >>> import time
  >>> for i in range(3):
❶         print('Tick')
❷         time.sleep(1)
```

```
❸         print('Tock')

❹         time.sleep(1)

  Tick
  Tock
  Tick
  Tock
  Tick
  Tock
❺ >>> time.sleep(5)
```

for循环将打印Tick❶，暂停一秒钟❷，打印Tock❸，暂停一秒钟❹，打印Tick，暂停，如此继续，直到Tick和Tock分别被打印3次。

time.sleep()函数将阻塞（也就是说，它不会返回或让程序执行其他代码），直到传递给time.sleep()的秒数流逝。例如，如果输入time.sleep(5) ❺，会在5秒后才看到下一个提示符（>>>）。

请注意，在IDLE中按Ctrl-C不会中断time.sleep()调用。IDLE会等待到暂停结束，再抛出KeyboardInterrupt异常。要绕过这个问题，不要用一次time.sleep(30)调用来暂停30秒，而是使用for循环执行30次time.sleep(1)调用。

```
>>> for i in range(30):

        time.sleep(1)
```

如果在这30秒内的某个时候按Ctrl-C，应该马上看到抛出KeyboardInterrupt异常。

15.2 数字四舍五入

在处理时间时，你会经常遇到小数点后有许多数字的浮点值。为了让这些值更易于处理，可以用Python内置的round()函数将它们缩短，该函数按照指定的精度四舍五入到一个浮点数。只要传入要舍入的数字，再加上可选的第二个参数，指明需要传入到小数点后多少位。如果省略第二个参数，round()将数字四舍五入到最接近的整数。在交互式环境中输入以下代码：

```
>>> import time

>>> now = time.time()

>>> now
```

```
1425064108.017826
>>> round(now, 2)
1425064108.02
>>> round(now, 4)
1425064108.0178
>>> round(now)
1425064108
```

导入time，将time.time()保存在now中之后，我们调用round(now, 2)，将now舍入到小数点后两位数字，round(now, 4)舍入到小数点后四位数字，round(now)舍入到最接近的整数。

15.3 项目：超级秒表

假设要记录在没有自动化的枯燥任务上花了多少时间。你没有物理秒表，要为笔记本或智能手机找到一个免费的秒表应用，没有广告，且不会将你的浏览历史发送给市场营销人员，又出乎意料地困难（在你同意的许可协议中，它说它可以这样做。你确实阅读了许可协议，不是吗？）。你可以自己用Python写一个简单的秒表程序。

总的来说，你的程序需要完成：

- 记录从按下回车键开始，每次按键的时间，每次按键都是一个新的“单圈”。
- 打印圈数、总时间和单圈时间。

这意味着代码将需要完成以下任务：

- 在程序开始时，通过调用time.time()得到当前时间，将它保存为一个时间戳。在每个单圈开始时也一样。
- 记录圈数，每次用户按下回车键时加1。
- 用时间戳相减，得到计算流逝的时间。
- 处理KeyboardInterrupt异常，这样用户可以按Ctrl-C退出。

打开一个新的文件编辑器窗口，并保存为stopwatch.py。

第1步：设置程序来记录时间

秒表程序需要用到当前时间，所以要导入的time模块。程序在调用input()之前，也应该向用户打印一些简短的说明，这样计时器可以在用户按下回车键后开始。然后，代码将开始记录单圈时间。在文件编辑器中输入以下代码，为其余的代码编写TODO注释，作为占位符：

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

# Display the program's instructions.
print('Press ENTER to begin. Afterwards, press ENTER to "click" the stopwat
Press Ctrl-C to quit.')
input()                  # press Enter to begin
print('Started.')
startTime = time.time()      # get the first lap's start time
lastTime = startTime
lapNum = 1

# TODO: Start tracking the lap times.
```

既然我们已经编码显示了用户说明，那就开始第一圈，记下时间，并将圈数设为1。

第2步：记录并打印单圈时间

现在，让我们编码开始每一个新的单圈，计算前一圈花了多少时间，并计算自启动秒表后经过的总时间。我们将显示的单圈时间和总时间，为每个新的单圈增加圈计数。将下面的代码添加到程序中：

```
#! python3
# stopwatch.py - A simple stopwatch program.

import time

--snip

--
# Start tracking the lap times.

❶ try:

❷     while True:

          input()
```

```
❸         lapTime = round(time.time() - lastTime, 2)

❹         totalTime = round(time.time() - startTime, 2)

❺         print('Lap #%s: %s (%s)' % (lapNum, totalTime, lapTime), end='')

        lapNum += 1

        lastTime = time.time() # reset the last lap time

❻ except KeyboardInterrupt:

    # Handle the Ctrl-C exception to keep its error message from displaying

    print('\nDone.')
```

如果用户按Ctrl-C停止秒表，KeyboardInterrupt异常将抛出，如果程序的执行不是一个try语句，就会崩溃。为了防止崩溃，我们将这部分程序包装在一个try语句中❶。我们将在except子句中处理异常❻，所以当Ctrl-C按下并引发异常时，程序执行转向except子句，打印Done，而不是KeyboardInterrupt错误消息。在此之前，执行处于一个无限循环中❷，调用input()并等待，直到用户按下回车键结束一圈。当一圈结束时，我们用当前时间time.time()减去该圈开始的时间lastTime，计算该圈花了多少时间❸。我们用当前时间减去秒表最开始启动的时间startTime，计算总共流逝的时间❹。

由于这些时间计算的结果在小数点后有许多位（如4.766272783279419），所以我们在❸和❹行用round()函数，将浮点值四舍五入到小数点后两位。

在❺行，我们打印出圈数，消耗的总时间和单圈时间。由于用户为input()调用按下回车时，会在屏幕上打印一个换行，所以我们向print()函数传入end='，避免输出重复空行。打印单圈信息后，我们将计数器lapNum加1，将lastTime设置为当前时间（这就是下一圈的开始时间），从而为下一圈做好准备。

第3步：类似程序的想法

时间追踪为程序打开了几种可能性。虽然可以下载应用程序来做其中一些事情，但自己编程的好处是它们是免费的，而且不会充斥着广告和无用的功能。可以编写类似的程序来完成以下任务：

- 创建一个简单的工时表应用程序，当输入一个人的名字时，用当前的时间记录下他们进入或离开的时间。
- 为你的程序添加一个功能，显示自一项处理开始以来的时间，诸如利用requests模块进行的下载（参见第11章）。
- 间歇性地检查程序已经运行了多久，并为用户提供了一个机会，取消耗时太长的任务。

15.4 datetime模块

time模块用于取得Unix纪元时间戳，并加以处理。但是，如果以更方便的格式显示日期，或对日期进行算术运算（例如，搞清楚205天前是什么日期，或123天后是什么日期），就应该使用datetime模块。

datetime模块有自己的datetime数据类型。datetime值表示一个特定的时刻。在交互式环境中输入以下代码：

```
  >>> import datetime
❶ >>> datetime.datetime.now()
❷ datetime.datetime(2015, 2, 27, 11, 10, 49, 55, 53)
❸ >>> dt = datetime.datetime(2015, 10, 21, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
  (2015, 10, 21)
   >>> dt.hour, dt.minute, dt.second
  (16, 29, 0)
```

调用datetime.datetime.now()❶返回一个datetime对象❷，表示当前的日期和时间，根据你的计算机的时钟。这个对象包含当前时刻的年、月、日、时、分、秒和微秒。也可以利用datetime.datetime()函数❸，向它传入代表年、月、日、时、分、秒的整数，得到特定时刻的datetime对象。这些整数将保存在datetime对象的year、month、day❹、hour、minute和second❺属性中。

Unix纪元时间戳可以通过datetime.datetime.fromtimestamp()，转换为datetime对象。datetime 对象的日期和时间将根据本地时区转换。在交互式环境中输入以下代码：

```
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2015, 2, 27, 11, 13, 0, 604980)
```

调用datetime.datetime.fromtimestamp()并传入1000000，返回一个datetime对象，表示Unix纪元后1000000秒的时刻。传入time.time()，即当前时刻的Unix纪元时间戳，则返回当前时刻的datetime对象。因此，表达式datetime.datetime.now()和datetime.datetime.fromtimestamp(time.time())做的事情相同，它们都返回当前时刻的datetime对象。

注意

这些例子是在一台设置了太平洋标准时间的计算机上输入的。如果你在另一个时区，结果会

有所不同。

datetime对象可以用比较操作符进行比较，弄清楚谁在前面。后面的datetime对象是“更大”的值。在交互式环境中输入以下代码：

```
❶ >>> halloween2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❷ >>> newyears2016 = datetime.datetime(2016, 1, 1, 0, 0, 0)
   >>> oct31_2015 = datetime.datetime(2015, 10, 31, 0, 0, 0)
❸ >>> halloween2015 == oct31_2015
   True
❹ >>> halloween2015 > newyears2016
   False
❺ >>> newyears2016 > halloween2015
   True
   >>> newyears2016 != oct31_2015
```

```
True
```

为2015年10月31日的第一个时刻（午夜）创建一个datetime对象，将它保存在halloween2015中❶。为2016年1月1日的第一个时刻创建一个datetime对象，将它保存在newyears2016中❷。然后，为2015年10月31日的午夜创建另一个对象，将它保存在oct31_2015中。比较halloween2015和oct31_2015，它们是相等的❸。比较newyears2016和halloween2015，newyears2016大于（晚于）halloween2015 ❹❺。

15.4.1 timedelta数据类型

datetime模块还提供了timedelta数据类型，它表示一段时间，而不是一个时刻。在交互式环境中输入以下代码：

```
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
  (11, 36548, 0)
  >>> delta.total_seconds()
  986948.0
  >>> str(delta)
  '11 days, 10:09:08'
```

要创建timedelta对象，就用datetime.timedelta()函数。datetime.timedelta()函数接受关键字参数weeks、days、hours、minutes、seconds、milliseconds和microseconds。没有month和year关键字参数，因为“月”和“年”是可变的时间，依赖于特定月份或年份。timedelta对象拥有的总时间以天、秒、微秒来表示。这些数字分别保存在days、seconds和microseconds属性中。total_seconds()方法返回只以秒表示的时间。将一个timedelta对象传入str()，将返回格式良好的、人类可读的字符串表示。

在这个例子中，我们将关键字参数传入datetime.delta()，指定11天、10小时、9分和8秒的时间，将返回的timedelta对象保存在delta中❶。该timedelta对象的days属性为11，seconds属性为36548（10小时、9分钟、8秒，以秒表示）❷。调用total_seconds()告诉我们，11天、10小时、9分和8秒是986948秒。最后，将这个timedelta对象传入str()，返回一个字符串，明确解释了这段时间。

算术运算符可以用于对datetime值进行日期运算。例如，要计算今天之后1000天的日期，在交互式环境中输入以下代码：

```
>>> dt = datetime.datetime.now()
>>> dt
datetime.datetime(2015, 2, 27, 18, 38, 50, 636181)
>>> thousandDays = datetime.timedelta(days=1000)
```

```
>>> dt + thousandDays

datetime.datetime(2017, 11, 23, 18, 38, 50, 636181)
```

首先，生成表示当前时刻的datetime对象，保存在dt中。然后生成一个timedelta对象，表示1000天，保存在thousandDays中。dt与thousandDays相加，得到一个datetime对象，表示现在之后的1000天。Python将完成日期运算，弄清楚2015年2月27日之后的1000天，将是2017年11月23日。这很有用，因为如果要从一个给定的日期计算1000天之后，需要记住每个月有多少天，闰年的因素和其他棘手的细节。datetime模块为你处理所有这些问题。

利用+和-运算符，timedelta对象与datetime对象或其他timedelta对象相加或相减。利用*和/运算符，timedelta对象可以乘以或除以整数或浮点数。在交互式环境中输入以下代码：

```
❶ >>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)

❷ >>> aboutThirtyYears = datetime.timedelta(days=365 * 30)

  >>> oct21st

  datetime.datetime(2015, 10, 21, 16, 29)
```

```
>>> oct21st - aboutThirtyYears
datetime.datetime(1985, 10, 28, 16, 29)
>>> oct21st - (2 * aboutThirtyYears)
datetime.datetime(1955, 11, 5, 16, 29)
```

这里，我们生成了一个DateTime对象，表示2015年10月21日❶，以及一个timedelta对象，表示大约30年的时间（我们假设每年为365天）❷。从oct21st中减去aboutThirtyYears，我们就得到一个datetime对象，表示2015年10月21日前30年的一天。从oct21st中减去2 * aboutThirtyYears，得到一个datetime对象，表示2015年10月21日之前60年的一天。

15.4.2 暂停直至特定日期

time.sleep()方法可以暂停程序若干秒。利用一个while循环，可以让程序暂停，直到一个特定的日期。例如，下面的代码会继续循环，直到2016年万圣节：

```
import datetime
import time
halloween2016 = datetime.datetime(2016, 10, 31, 0, 0, 0)
while datetime.datetime.now() <  halloween2016:
    time.sleep(1)
```

time.sleep(1)调用将暂停你的Python程序，这样计算机不会浪费CPU

处理周期，一遍又一遍地检查时间。相反，while循环只是每秒检查一次，在2016年万圣节（或你编程让它停止的时间）后继续执行后面的程序。

15.4.3 将datetime对象转换为字符串

Unix纪元时间戳和datetime对象对人类来说都不是很友好可读。利用strftime()方法，可以将datetime对象显示为字符串。（strftime()函数名中的f表示格式，format）。

该的strftime()方法使用的指令类似于Python的字符串格式化。表15-1列出了完整的strftime()指令。

表**15-1 strftime()**指令

strftime指令	含义
%Y	带世纪的年份，例如'2014'
%y	不带世纪的年份，'00'至'99'（1970至2069）
%m	数字表示的月份, '01'至'12'
%B	完整的月份，例如'November'
%b	简写的月份，例如'Nov'
%d	一月中的第几天，'01'至'31'
%j	一年中的第几天，'001'至'366'
%w	一周中的第几天，'0'（周日）至'6'（周六）

%A	完整的周几，例如'Monday'
%a	简写的周几，例如'Mon'
%H	小时（24小时时钟），'00'至'23'
%I	小时（12小时时钟），'01'至'12'
%M	分，'00'至'59'
%S	秒，'00'至'59'
%p	'AM'或'PM'
%%	就是'%'字符

向strftime()传入一个定制的格式字符串，其中包含格式化指定（以及任何需要的斜线、冒号等），strftime()将返回一个格式化的字符串，表示datetime对象的信息。在交互式环境中输入以下代码：

```
>>> oct21st = datetime.datetime(2015, 10, 21, 16, 29, 0)
>>> oct21st.strftime('%Y/%m/%d %H:%M:%S')
'2015/10/21 16:29:00'
>>> oct21st.strftime('%I:%M %p')
```

```
'04:29 PM'
>>> oct21st.strftime("%B of '%y")
"October of '15"
```

这里，我们有一个datetime对象，表示2015年10月21日下午4点29分，保存在oct21st中。向strftime()传入定制的格式字符串'%Y/%m/%d %H:%M:%S，返回一个字符串，包含以斜杠分隔的2015、10和21，以冒号分隔的16、29和00。传入'%I:%M% p'则返回'04:29 PM'，传入"%B of '%y"则返回"October of '15"。请注意，strftime()不是以datetime.datetime开始。

15.4.4 将字符串转换成datetime对象

如果有一个字符串的日期信息，如'2015/10/21 16:29:00'或'October 21, 2015'，需要将它转换为datetime对象，就用datetime.datetime.strftime()函数。strptime()函数与strftime()方法相反。定制的格式字符串使用相同的指令，像strftime()一样。必须将格式字符串传入strptime()，这样它就知道如何解析和理解日期字符串（strptime()函数名中p表示解析，parse）。

在交互式环境中输入以下代码：

```
❶ >>> datetime.datetime.strptime('October 21, 2015', '%B %d, %Y')
  datetime.datetime(2015, 10, 21, 0, 0)
  >>> datetime.datetime.strptime('2015/10/21 16:29:00', '%Y/%m/%d %H:%M:%S'
```

```
datetime.datetime(2015, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '15", "%B of '%y")
datetime.datetime(2015, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
```

要从字符串'October 21, 2015'取得一个datetime对象，将'October 21, 2015'作为第一个参数传递给strptime()，并将对应于'October 21, 2015' 的定制格式字符串作为第二个参数❶。带有日期信息的字符串必须准确匹配定制的格式字符串，否则Python将抛出ValueError异常。

15.5 回顾**Python**的时间函数

在Python中，日期和时间可能涉及好几种不同的数据类型和函数。下面回顾了表示时间的3种不同类型的值：

- Unix纪元时间戳（time模块中使用）是一个浮点值或整型值，表示自1970年1月1日午夜0点（UTC）以来的秒数。
- datetime对象（属于datetime模块）包含一些整型值，保存在year、month、day、hour、minute和second等属性中。
- timedelta对象（属于datetime模块）表示的一段时间，而不是一个特定的时刻。

下面回顾了时间函数及其参数和返回值：

- time.time()函数返回一个浮点值，表示当前时刻的Unix纪元时间戳。
- time.sleep(seconds)函数让程序暂停seconds参数指定的秒数。
- datetime.datetime(year, month, day, hour, minute, second)函数返回参数指定的时刻的datetime对象。如果没有提供hour、minute或second参数，它们默认为0。
- datetime.datetime.now()函数返回当前时刻的datetime对象。
- datetime.datetime.fromtimestamp(epoch)函数返回epoch时间戳参数表示的时刻的datetime对象。
- datetime.timedelta(weeks, days, hours, minutes, seconds, milliseconds, microseconds)函数返回一个表示一段时间的timedelta对象。该函数的关键字参数都是可选的，不包括month或year。
- total_seconds()方法用于timedelta对象，返回timedelta对象表示的秒数。
- strftime(format)方法返回一个字符串，用format字符串中的定制格式来表示datetime对象表示的时间。详细格式参见表15-1。
- datetime.datetime.strptime(time_string, format)函数返回一个datetime对象，它的时刻由time_string指定，利用format字符串参数来解析。详细格式参见表15-1。

15.6 多线程

为了引入多线程的概念，让我们来看一个例子。假设你想安排一些代码，在一段延迟后或在特定时间运行。可以在程序启动时添加如下代码：

```
import time, datetime

startTime = datetime.datetime(2029, 10, 31, 0, 0, 0)
while datetime.datetime.now() <  startTime:
    time.sleep(1)

print('Program now starting on Halloween 2029')
--snip





--
```

这段代码指定2029年10月31日作为开始时间，不断调用time.sleep(1)，直到开始时间。在等待time.sleep()的循环调用完成时，程序不能做任何事情，它只是坐在那里，直到2029年万圣节。这是因为Python程序在默认情况下，只有一个执行线程。

要理解什么是执行线程，就要回忆第2章关于控制流的讨论，当时你想象程序的执行就像把手指放在一行代码上，然后移动到下一行，或是流控制语句让它去的任何地方。单线程程序只有一个“手指”。但多线程的程序有多个“手指”。每个“手指”仍然移动到控制流语句定义的下一行代码，但这些“手指”可以在程序的不同地方，同时执行不同的代码行（到目前为止，本书所有的程序一直是单线程的）。

不必让所有的代码等待，直到time.sleep()函数完成，你可以使用Python的threading模块，在单独的线程中执行延迟或安排的代码。这个单独的线程将因为time.sleep()调用而暂停。同时，程序可以在原来的线程中做其他工作。

要得到单独的线程，首先要调用threading.Thread()函数，生成一个Thread对象。在新的文件中输入以下代码，并保存为threadDemo.py：

```
  import threading, time
  print('Start of program.')

❶ def takeANap():
      time.sleep(5)
      print('Wake up!')

❷ threadObj = threading.Thread(target=takeANap)
❸ threadObj.start()

  print('End of program.')
```

在❶行，我们定义了一个函数，希望用于新线程中。为了创建一个Thread对象，我们调用threading.Thread()，并传入关键字参数target=takeANap❷。这意味着我们要在新线程中调用的函数是takeANap()。请注意，关键字参数是target=takeANap，而不是target=takeANap()。这是因为你想将takeANap()函数本身作为参数，而不是调用takeANap()，并传入它的返回值。

我们将threading.Thread()创建的Thread对象保存在threadObj中，然后调用threadObj.start()❸，创建新的线程，并开始在新线程中执行目标函数。如果运行该程序，输出将像这样：

```
Start of program.
End of program.
Wake up!
```

这可能有点令人困惑。如果print('End of program.')是程序的最后一行，你可能会认为，它应该是最后打印的内容。Wake up!在它后面是因为，当threadObj.start()被调用时，threadObj的目标函数运行在一个新的执行线程中。将它看成是第二根“手指”，出现在takeANap()函数开始处。主线程继续print('End of program.')。同时，新线程已执行了time.sleep(5)调用，暂停5秒钟。之后它从5秒钟小睡中醒来，打印了'Wake up!'，然后从takeANap()函数返回。按时间顺序，'Wake up!'是程序最后打印的内容。

通常，程序在文件中最后一行代码执行后终止（或调用sys.exit()）。但threadDemo.py有两个线程。第一个是最初的线程，从程序开始处开始，在print('End of program.')后结束。第二个线程是调用threadObj.start()时创建的，始于takeANap()函数的开始处，在takeANap()返回后结束。

在程序的所有线程终止之前，Python程序不会终止。在运行threadDemo.py时，即使最初的线程已经终止，第二个线程仍然执行time.sleep(5)调用。

15.6.1 向线程的目标函数传递参数

如果想在新线程中运行的目标函数有参数，可以将目标函数的参数传入threading.Thread()。例如，假设想在自己的线程中运行以下print()调用：

```
>>> print('Cats', 'Dogs', 'Frogs', sep=' & ')

Cats & Dogs & Frogs
```

该print()调用有3个常规参数：'Cats'、 'Dogs'和'Frogs'，以及一个关键字参数：sep=' & '。常规参数可以作为一个列表，传递给threading.Thread()中的args关键字参数。关键字参数可以作为一个字典，传递给threading.Thread()中的kwargs关键字参数。

在交互式环境中输入以下代码：

```
>>> import threading

>>> threadObj = threading.Thread(target=print, args=['Cats', 'Dogs', 'Frogs

kwargs={'sep': ' & '})

>>> threadObj.start()
```

```
Cats & Dogs & Frogs
```

为了确保参数'Cats'、'Dogs'和'Frogs'传递给新线程中的print()，我们将args=['Cats', 'Dogs', 'Frogs']传入threading.Thread()。为了确保关键字参数sep=' & '传递给新线程中的print()，我们将kwargs={'sep': '& '}传入threading.Thread()。

threadObj.start()调用将创建一个新线程来调用print()函数，它会传入'Cats'、'Dogs'和'Frogs'作为参数，以及' & '作为sep关键字参数。

下面创建新线程调用print()的方法是不正确的：

```
threadObj = threading.Thread(target=print('Cats', 'Dogs', 'Frogs', sep=' &
```

这行代码最终会调用print()函数，将它的返回值（print()的返回值总是无）作为target关键字参数。它没有传递print()函数本身。如果要向新线程中的函数传递参数，就使用threading.Thread()函数的args和kwargs关键字参数。

15.6.2 并发问题

可以轻松地创建多个新线程，让它们同时运行。但多线程也可能会导致所谓的并发问题。如果这些线程同时读写变量，导致互相干扰，就会发生并发问题。并发问题可能很难一致地重现，所以难以调试。

多线程编程本身就是一个广泛的主题，超出了本书的范围。必须记住的是：为了避免并发问题，绝不让多个线程读取或写入相同的变量。当创建一个新的Thread对象时，要确保其目标函数只使用该函数中的局部变量。这将避免程序中难以调试的并发问题。

注意

在http://nostarch.com/automatestuff/，有关于多线程编程的初学者教程。

15.7 项目：多线程XKCD下载程序

在第11章，你编写了一个程序，从XKCD网站下载所有的XKCD漫画。这是一个单线程程序：它一次下载一幅漫画。程序运行的大部分时间，都用于建立网络连接来开始下载，以及将下载的图像写入硬盘。如果你有宽带因特网连接，单线程程序并没有充分利用可用的带宽。

多线程程序中有一些线程在下载漫画，同时另一些线程在建立连接，或将漫画图像文件写入硬盘。它更有效地使用Internet连接，更迅速地下载这些漫画。打开一个新的文件编辑器窗口，并保存为multidownloadXkcd.py。你将修改这个程序，添加多线程。经过全面修改的源代码可从http://nostarch.com/automatestuff/下载。

第1步：修改程序以使用函数

该程序大部分是来自第 11 章的相同下载代码，所以我会跳过Requests 和BeautifulSoup代码的解释。需要完成的主要变更是导入threading模块，并定义downloadXkcd()函数，该函数接受开始和结束的漫画编号作为参数。

例如，调用downloadXkcd(140，280)将循环执行下载代码，下载漫画http://xkcd. com/140 、http://xkcd.com/141、http://xkcd.com/142等，直到http://xkcd.com/279。你创建的每个线程都会调用downloadXkcd()，并传入不同范围的漫画进行下载。

将下面的代码添加到multidownloadXkcd.py程序中：

```
  #! python3
  # multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

  import requests, os, bs4, threading
❶ os.makedirs('xkcd', exist_ok=True)      # store comics in ./xkcd

❷ def downloadXkcd(startComic, endComic):
❸     for urlNumber in range(startComic, endComic):
          # Download the page.
```

```
        print('Downloading page http://xkcd.com/%s...' % (urlNumber))
❹       res = requests.get('http://xkcd.com/%s' % (urlNumber))
        res.raise_for_status()

❺       soup = bs4.BeautifulSoup(res.text)

        # Find the URL of the comic image.
❻       comicElem = soup.select('#comic img')
        if comicElem == []:
            print('Could not find comic image.')
        else:
❼           comicUrl = comicElem[0].get('src')
            # Download the image.
            print('Downloading image %s...' % (comicUrl))
❽           res = requests.get(comicUrl)
            res.raise_for_status()

            # Save the image to ./xkcd.
            imageFile = open(os.path.join('xkcd', os.path.basename(comicU
            for chunk in res.iter_content(100000):
                imageFile.write(chunk)
            imageFile.close()

# TODO: Create and start the Thread objects.
# TODO: Wait for all threads to end.
```

导入需要的模块后，❶行创建了一个目录来保存漫画，并开始定义downloadxkcd()❷。循环遍历指定范围中的所有编号❸，并下载每个页面❹。用Beautiful Soup查看每一页的HTML❺，找到漫画图像❻。如果页面上没有的漫画图像，就打印一条消息。否则，取得图片的URL❼，并下载图像❽。最后，将图像保存到我们创建的目录中。

第2步：创建并启动线程

既然已经定义 downloadXkcd()，我们将创建多个线程，每个线程调用downloadXkcd()，从 XKCD 网站下载不同范围的漫画。将下面的代码添加到multidownloadXkcd.py中，放在downloadXkcd()函数定义之后：

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.
```

```
--snip

--
# Create and start the Thread objects.

downloadThreads = []               # a list of all the Thread objects

for i in range(0, 1400, 100):       # loops 14 times, creates 14 threads

    downloadThread = threading.Thread(target=downloadXkcd, args=(i, i + 99)

    downloadThreads.append(downloadThread)

    downloadThread.start()
```

首先，我们创建了一个空列表downloadThreads，该列表帮助我们追踪创建的多个Thread对象。然后开始for循环。在每次循环中，我们利用threading.Thread()创建一个Thread对象，将它追加到列表中，并调用start()，开始在新线程中运行downloadXkcd()。因为for循环将变量i设置为从0到1400，步长为100，所以i在第一次迭代时为0，第二次迭代时为100，第三次为200，以此类推。因为我们将args=(I, I+99)传递给threading.Thread()，所以在第一次迭代时，传递给downloadXkcd()的两个参数将是0和99，第二次迭代是100和199，第三次是200和299，以次类推。

由于调用了Thread对象的start()方法，新的线程开始运行downloadXkcd()中的代码，主线程将继续for循环的下一次迭代，创造下一个线程。

第3步：等待所有线程结束

主线程正常执行，同时我们创建的其他线程下载漫画。但是假定主线程中有一些代码，你希望所有下载线程完成后再执行。调用Thread对象join()方法将阻塞，直到该线程完成。利用一个for循环，遍历downloadThreads列表中的所有Thread对象，主线程可以调用其他每个线程的join()方法。将以下代码添加到程序的末尾：

```
#! python3
# multidownloadXkcd.py - Downloads XKCD comics using multiple threads.

--snip

--

# Wait for all threads to end.

for downloadThread in downloadThreads:
```

```
    downloadThread.join()

print('Done.')
```

所有的join()调用返回后，'Done.'字符串才会打印，如果一个Thread对象已经完成，那么调用它的join()方法时，该方法就会立即返回。如果想扩展这个程序，添加一些代码，在所有漫画下载后运行，就可以用新的代码替换print('Done.')。

15.8 从Python启动其他程序

利用内建的subprocess模块中的Popen()函数，Python程序可以启动计算机中的其他程序（Popen()函数名中的P表示process，进程）。如果你打开了一个应用程序的多个实例，每个实例都是同一个程序的不同进程。例如，如果你同时打开了Web浏览器的多个窗口，每个窗口都是Web浏览器程序的不同进程。参见图15-1，这是同时打开多个计算器进程的例子。

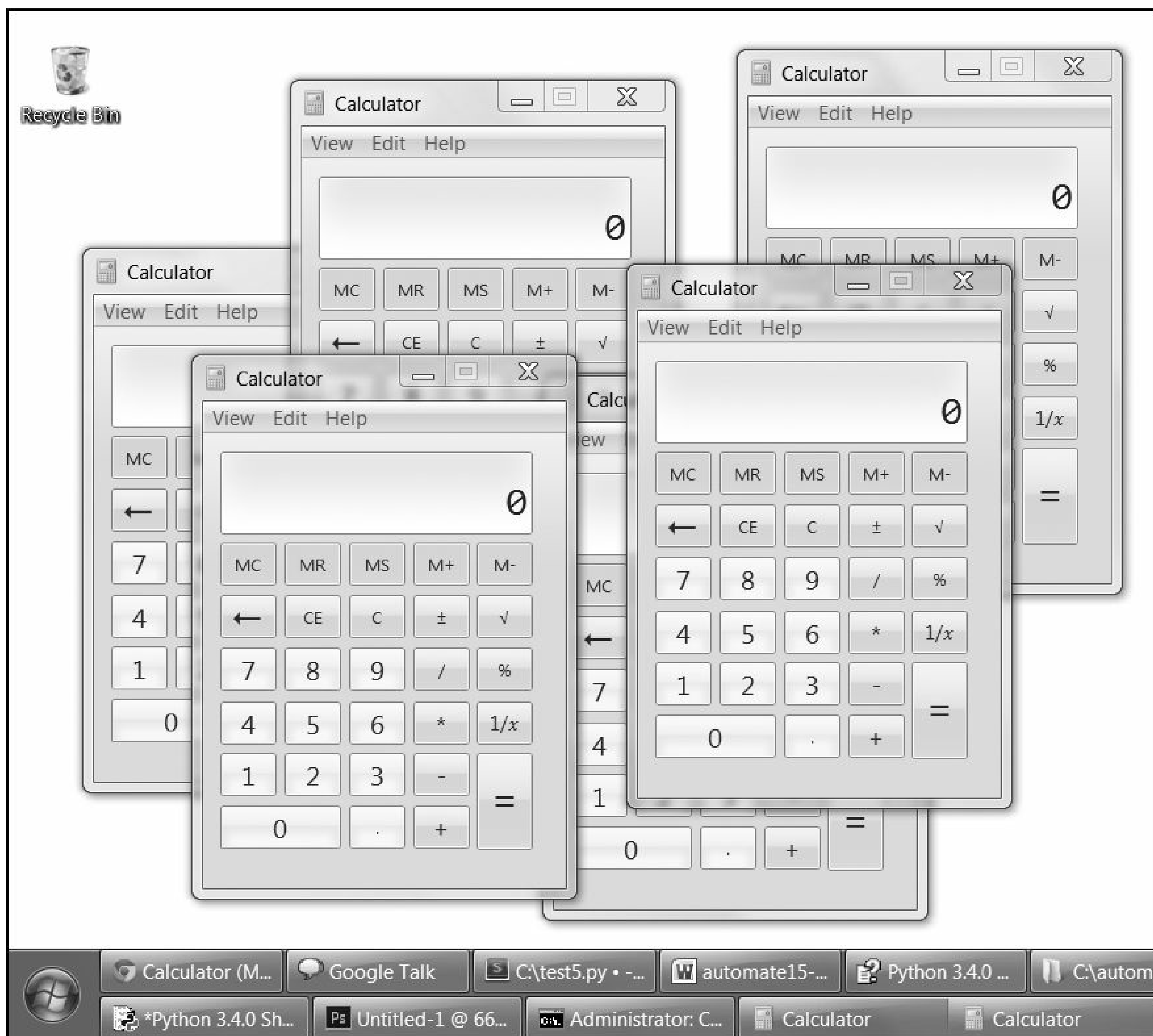


图15-1 相同的计算器程序，六个正在运行的进程

每个进程可以有多个线程。不像线程，进程无法直接读写另一个进程的变量。如果你认为多线程程序是多个手指在追踪源代码，那么同一个程序打开多个进程就像有一个朋友拿着程序源代码的独立副本。你们都独立地执行相同的程序。

如果想在Python脚本中启动一个外部程序，就将该程序的文件名传递给subprocess.Popen()（在Windows中，右键点击该应用程序的开始菜单项，然后选择“属性”，查看应用程序的文件名。在OS X上，按住Ctrl键单击该应用程序并选择“显示包内容”，找到可执行文件的路径）。Popen()函数随后将立即返回。请记住，启动的程序和你的Python程序不

在同一线程中运行。

在Windows计算机上，在交互式环境中输入以下代码：

```
>>> import subprocess

>>> subprocess.Popen('C:\\Windows\\System32\\calc.exe')

< subprocess.Popen object at 0x0000000003055A58>
```

在Ubuntu Linux上，可以输入以下代码：

```
>>> import subprocess

>>> subprocess.Popen('/usr/bin/gnome-calculator')

< subprocess.Popen object at 0x7f2bcf93b20>
```

在OS X上，过程稍有不同。参见15.8.5节“用默认应用程序打开文件”。

返回值是一个Popen对象，它有两个有用的方法：poll()和wait()。

可以认为poll()方法是问你的朋友，她是否执行完毕你给她的代码。如果这个进程在poll()调用时仍在运行，poll()方法就返回None。如果该程序已经终止，它会返回该进程的整数退出代码。退出代码用于说明进程是无错终止（退出代码为0），还是一个错误导致进程终止（退出代码非零，通常为1，但可能根据程序而不同）。

wait()方法就像是等着你的朋友执行完她的代码，然后你继续执行你的代码。wait()方法将阻塞，直到启动的进程终止。如果你希望你的程序暂停，直到用户完成与其他程序，这非常有用。wait()的返回值是进程的整数退出代码。

在Windows上，在交互环境中输入以下代码。请注意， wait()的调用将阻塞，直到退出启动的计算器程序。

```
❶ >>> calcProc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')
❷ >>> calcProc.poll() == None
  True
❸ >>> calcProc.wait()
  0
  >>> calcProc.poll()
  0
```

这里，我们打开了计算器程序❶。在它仍在运行时，我们检查poll()是否返回None❷。它应该返回None，因为该进程仍在运行。然后，我们关闭计算器程序，并对已终止的进程调用wait()❸。wait()和poll()现在返回0，说明该进程终止且无错。

15.8.1 向Popen()传递命令行参数

用Popen()创建进程时，可以向进程传递命令行参数。要做到这一点，向Popen()传递一个列表，作为唯一的参数。该列表中的第一个字符串是要启动的程序的可执行文件名，所有后续的字符串将是该程序启动时，传递给该程序的命令行参数。实际上，这个列表将作为被启动程序的sys.argv的值。

大多数具有图形用户界面（GUI）的应用程序，不像基于命令行或基于终端的程序那样尽可能地使用命令行参数。但大多数GUI应用程序将接受一个参数，表示应用程序启动时立即打开的文件。例如，如果你使用的是Windows，创建一个简单的文本文件C:\hello.txt，然后在交互式环境中输入以下代码：

```
>>> subprocess.Popen(['C:\\Windows\\notepad.exe', 'C:\\hello.txt'])

< subprocess.Popen object at 0x00000000032DCEB8>
```

这不仅会启动记事本应用程序，也会让它立即打开C:\hello.txt。

15.8.2 Task Scheduler、launchd和cron

如果你精通计算机，可能知道 Windows 上的 Task Scheduler，OS X 上的launchd，或Linux上的cron调度程序。这些工具文档齐全，而且可靠，它们都允许你安排应用程序在特定的时间启动。如果想更多地了解它们，可以在http://nostarch. com/automatestuff/ 找到教程的链接。

利用操作系统内置的调度程序，你不必自己写时钟检查代码来安排你的程序。但是，如果只需要程序稍作停顿，就用time.sleep()函数。或者不使用操作系统的调度程序，代码可以循环直到特定的日期和时间，每次循环时调用time.sleep(1)。

15.8.3 用**Python**打开网站

webbrowser.open()函数可以从程序启动Web浏览器，打开指定的网站，而不是用subprocess.Popen()打开浏览器应用程序。详细内容参见第11章的“项目：利用webbrowser模块的mapIt.py”一节。

15.8.4 运行其他**Python**脚本

可以在Python中启动另一个Python脚本，就像任何其他的应用程序一样。只需向Popen()传入python.exe可执行文件，并将想运行的.py脚本的文件名作为它的参数。例如，下面代码将运行第1章的hello.py脚本：

```
>>> subprocess.Popen(['C:\\python34\\python.exe', 'hello.py'])

< subprocess.Popen object at 0x000000000331CF28>
```

向Popen()传入一个列表，其中包含Python可执行文件的路径字符串，以及脚本文件名的字符串。如果要启动的脚本需要命令行参数，就将它们添加列表中，放在脚本文件名后面。在Windows上，Python可执行文件的路径是C：\python34\ python.exe。在OS X上，是/Library/Frameworks/Python.framework/ Versions/3.3/bin/python3。在

Linux上，是/usr/bin/python3。

不同于将Python程序导入为一个模块，如果Python程序启动了另一个Python程序，两者将在独立的进程中运行，不能分享彼此的变量。

15.8.5 用默认的应用程序打开文件

双击计算机上的.txt文件，会自动启动与.txt文件扩展名关联的应用程序。计算机上已经设置了一些这样的文件扩展名关联。利用Popen()，Python也可以用这种方式打开文件。

每个操作系统都有一个程序，其行为等价于双击文档文件来打开它。在Windows上，这是start程序。在OS X上，这是open程序。在Ubuntu Linux上，这是see程序。在交互式环境中输入以下代码，根据操作系统，向Popen()传入'start'、'open'或'see'：

```
>>> fileObj = open('hello.txt', 'w')
>>> fileObj.write('Hello world!')
12
>>> fileObj.close()
>>> import subprocess
>>> subprocess.Popen(['start
```

```
', 'hello.txt'], shell=True)
```

这里，我们将Hello world!写入一个新的hello.txt文件。然后调用Popen()，传入一个列表，其中包含程序名称（在这个例子中，是Windows上的'start'），以及文件名。我们也传入了shell=True关键字参数，这只在Windows上需要。操作系统知道所有的文件关联，能弄清楚应该启动哪个程序，比如Notepad.exe，来处理hello.txt文件。

在OS X上，open程序用于打开文档文件和程序。如果你有Mac，在交互式环境中输入以下代码：

```
>>> subprocess.Popen(['open', '/Applications/Calculator.app/'])

< subprocess.Popen object at 0x10202ff98>
```

计算器应用程序应该会打开。

Unix哲学

程序精心设计，能被其他程序启动，这样的程序比单独使用它们自己的代码更强大。Unix的哲学是一组由UNIX操作系统（现代的Linux和OS X也是基于它）的程序员建立的软件设计原则。它认为：编写小的、目的有限的、能互操作的程序，胜过大的、功能丰富的应用程序。

> 较小的程序更容易理解，通过能够互操作，它们可以是更强大的应用程序的构建块。智能手机应用程序也遵循这种方式。如果你的餐厅应用程序需要显示一间咖啡店的方位，开发者不必重新发明轮子，编写自己的地图代码。餐厅应用程序只是启动一个地图应用程序，同时传入咖啡店的地址，就像Python代码调用一个函数，并传入参数一样。
>
> 你在本书中编写的Python程序大多符合Unix哲学，尤其是在一个重要的方面：它们使用命令行参数，而不是input()函数调用。如果程序需要的所有信息都可以事先提供，最好是用命令行参数传入这些信息，而不是等待用户键入它。这样，命令行参数可以由人类用户键入，也可以由另一个程序提供。这种互操作的方式，让你的程序可以作为另一个程序的部分而复用。
>
> 唯一的例外是，你不希望口令作为命令行参数传入，因为命令行可能记录它们，作为命令历史功能的一部分。在需要输入口令时，程序应该调用input()函数。
>
> 在https://en.wikipedia.org/wiki/Unix_philosophy/，你可以阅读更多有关Unix哲学的内容。

15.9 项目：简单的倒计时程序

就像很难找到一个简单的秒表应用程序一样，也很难找到一个简单的倒计时程序。让我们来写一个倒计时程序，在倒计时结束时报警。

总的来说，程序要做到：

- 从60倒数。
- 倒数至0时播放声音文件（alarm.wav）。

这意味着代码将需要做到以下几点：

- 在显示倒计时的每个数字之间，调用time.sleep()暂停一秒。
- 调用subprocess.Popen()，用默认的应用程序播放声音文件。

打开一个新的文件编辑器窗口，并保存为countdown.py。

第1步：倒计时

这个程序需要time模块的time.sleep()函数，subprocess模块的subprocess. Popen()函数。输入以下代码并保存为countdown.py：

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess
```

```
❶ timeLeft = 60
  while timeLeft > 0:
❷     print(timeLeft, end='')
❸     time.sleep(1)
❹     timeLeft = timeLeft - 1

  # TODO: At the end of the countdown, play a sound file.
```

导入time和subprocess后，创建变量timeleft，保存倒计时剩下的秒数❶。它从60开始，或者可以根据需要更改这里的值，甚至通过命令行参数设置它。

在while循环中，显示剩余次数❷，暂停一秒钟❸，再减少timeleft变量的值❹，然后循环再次开始。只要timeleft大于0，循环就继续。在这之后，倒计时就结束了。

第2步：播放声音文件

虽然有第三方模块，播放各种声音文件，但快速而简单的方法，是启动用户使用的任何播放声音文件的应用程序。操作系统通过.wav文件扩展名，会弄清楚应该启动哪个应用程序来播放该文件。这个.wav文件很容易变成其他声音文件格式，如.mp3或.ogg。

可以使用计算机上的任何声音文件，在倒计时结束播放，也可以从http://nostarch. com/automatestuff/下载alarm.wav 。

在程序中添加以下代码：

```
#! python3
# countdown.py - A simple countdown script.

import time, subprocess

--snip
```

```
--
# At the end of the countdown, play a sound file.



subprocess.Popen(['start', 'alarm.wav'], shell=True)
```

while循环结束后，alarm.wav（或你选择的声音文件）将播放，通知用户倒计时结束。在Windows上，要确保传入Popen()的列表中包含'start'，并传入关键字参数shell=True。在OS X上，传入'open'，而不是'start’，并去掉shell=True。

除了播放声音文件之外，你可以在一个文本文件中保存一条消息，例如Break time is over!。然后在倒计时结束时用Popen()打开它。这实际上创建了一个带消息的弹出窗口。或者你可以在倒计时结束时，用webbrowser.open()函数打开特定网站。不像在网上找到的一些免费倒计时应用程序，你自己的倒计时程序的警报可以是任何你希望的方式！

第3步：类似程序的想法

倒计时是简单的延时，然后继续执行程序。这也可以用于其他应用程序和功能，诸如：

- 利用time.sleep()给用户一个机会，按下Ctrl-C取消的操作，例如删除文件。你的程序可以打印“Press Ctrl-C to cancel”，然后用try和except语句处理所有KeyboardInterrupt异常。
- 对于长期的倒计时，可以用timedelta对象来测量直到未来某个时间点（生日？周年纪念？）的天、时、分和秒数。

15.10 小结

对于许多编程语言，包括Python，Unix纪元（1970年1月1日午夜，UTC）是一个标准的参考时间。虽然time.time()函数模块返回一个Unix纪元时间戳（也就是自Unix纪元以来的秒数的浮点值），但datetime模块更适合执行日期计算、格式化和解析日期信息的字符串。

time.sleep()函数将阻塞（即不返回）若干秒。它可以用于在程序中暂停。但如果想安排程序在特定时间启动，http://nostarch.com/automatestuff/上的指南可以告诉你如何使用操作系统已经提供的调度程序。

threading 模块用于创建多个线程，如果需要下载多个文件或同时执行其他任务，这非常有用。但是要确保线程只读写局部变量，否则可能会遇到并发问题。

最后，Python程序可以用subprocess.Popen()函数，启动其他应用程序。命令行参数可以传递给Popen()调用，用该应用程序打开特定的文档。另外，也可以用Popen()启动start、open或see程序，利用计算机的文件关联，自动弄清楚用来打开文件的应用程序。通过利用计算机上的其他应用程序，Python程序可以利用它们的能力，满足你的自动化需求。

15.11 习题

1．什么是Unix纪元？

2．什么函数返回自Unix纪元以来的秒数？

3．如何让程序刚好暂停5秒？

4．round()函数返回什么？

5．datetime对象和timedelta对象之间的区别是什么？

6．假设你有一个函数名为spam()。如何在一个独立的线程中调用该函数并运行其中的代码？

7．为了避免多线程的并发问题，应该怎样做？

8．如何让Python程序运行C:\ Windows\ System32文件夹中的calc.exe程序？

15.12 实践项目

作为实践，编程完成下列任务。

15.12.1 美化的秒表

扩展本章的秒表项目，让它利用rjust()和ljust()字符串方法来“美化”的输出。（这些方法在第6章中介绍过）。输出不是像这样：

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

…而是像这样：

```
Lap # 1:   3.56 ( 3.56)
Lap # 2:   8.63 ( 5.07)
Lap # 3:  17.68 ( 9.05)
Lap # 4:  19.11 ( 1.43)
```

请注意，对于lapNum、lapTime和totalTime等整型和浮点型变量，你需要字符串版本，以便对它们调用字符串方法。接下来，利用第6章中介绍的pyperclip模块，将文本输出复制到剪贴板，以便用户可以将输出快速粘贴到一个文本文件或电子邮件中。

15.12.2 计划的Web漫画下载

编写一个程序，检查几个Web漫画的网站，如果自该程序上次访问以来，漫画有更新，就自动下载。操作系统的调度程序（Windows上的Task Scheduler，OS X上的launchd，以及Linux上的cron）可以每天运行你的Python程序一次。Python程序本身可以下载漫画，然后将它复制到桌面上，这样很容易找到。你就不必自己查看网站是否有更新（在http://nostarch.com/automatestuff/上有一份Web漫画的列表）。

第16章　发送电子邮件和短信

检查和答复电子邮件会占用大量的时间。当然，你不能只写一个程序来处理所有电子邮件，因为每个消息都需要有自己的回应。但是，一旦知道怎么编写收发电子邮件的程序，就可以自动化大量与电子邮件相关的任务。

例如，也许你有一个电子表格，包含许多客户记录，希望根据他们的年龄和位置信息，向每个客户发送不同格式的邮件。商业软件可能无法做这一点。好在，可以编写自己的程序来发送这些电子邮件，节省了大量复制和粘贴电子邮件的时间。

也可以编程发送电子邮件和短信，即使你远离计算机时，也能通知你。如果要自动化的任务需要执行几个小时，你不希望每过几分钟就回到计算机旁边，检查程序的状态。相反，程序可以在完成时向手机发短信，让你在离开计算机时，能专注于更重要的事情。

16.1 SMTP

正如HTTP是计算机用来通过因特网发送网页的协议，简单邮件传输协议（SMTP）是用于发送电子邮件的协议。SMTP 规定电子邮件应该如何格式化、加密、在邮件服务器之间传递，以及在你点击发送后，计算机要处理的所有其他细节。但是，你并不需要知道这些技术细节，因为Python的smtplib模块将它们简化成几个函数。

SMTP只负责向别人发送电子邮件。另一个协议，名为IMAP，负责取回发送给你的电子邮件，在16.3节“IMAP”中介绍。

16.2 发送电子邮件

你可能对发送电子邮件很熟悉，通过Outlook、Thunderbird或某个网站，如Gmail或雅虎邮箱。遗憾的是，Python没有像这些服务一样提供一个漂亮的图形用户界面。作为替代，你调用函数来执行SMTP的每个重要步骤，就像下面的交互式环境的例子。

注意

不要在IDLE中输入这个例子，因为smtp.example.com、bob@example.com、MY_SECRET_PASSWORD和alice@example.com只是占位符。这段代码仅仅勾勒出Python发送电子邮件的过程。

```
>>> import smtplib
>>> smtpObj = smtplib.SMTP('smtp.example.com', 587)
>>> smtpObj.ehlo()
(250, b'mx.example.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
>>> smtpObj.login('bob@example.com', 'MY_SECRET_PASSWORD
')
(235, b'2.7.0 Accepted')
>>> smtpObj.sendmail('bob@example.com', 'alice@example.com', 'Subject: So
```

```
long.\nDear Alice, so long and thanks for all the fish. Sincerely, Bob')

{}
>>> smtpObj.quit()

(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

在下面的小节中，我们将探讨每一步，用你的信息替换占位符，连接并登录到SMTP服务器，发送电子邮件，并从服务器断开连接。

16.2.1 连接到SMTP服务器

如果你曾设置了Thunderbird、Outlook或其他程序，连接到你的电子邮件账户，你可能熟悉配置SMTP服务器和端口。这些设置因电子邮件提供商而不同，但在网上搜索“< 你的提供商> SMTP设置”，应该能找到相应的服务器和端口。

SMTP服务器的域名通常是电子邮件提供商的域名，前面加上SMTP。例如，Gmail的 SMTP 服务器是smtp.gmail.com。表 16-1 列出了一些常见的电子邮件提供商及其SMTP服务器（端口是一个整数值，几乎总是587，该端口由命令加密标准TLS使用）。

表**16-1** 电子邮件提供商及其**SMTP**服务器

提供商	**SMTP**服务器域名

Gmail	*smtp.gmail.com*
Outlook.com/Hotmail.com	*smtp-mail.outlook.com*
Yahoo Mail	*smtp.mail.yahoo.com*
AT&T	*smpt.mail.att.net* (port 465)
Comcast	*smtp.comcast.net*
Verizon	*smtp.verizon.net* (port 465)

得到电子邮件提供商的域名和端口信息后，调用smtplib.SMTP()创建一个SMTP对象，传入域名作为一个字符串参数，传入端口作为整数参数。SMTP对象表示与SMTP邮件服务器的连接，它有一些发送电子邮件的方法。例如，下面的调用创建了一个SMTP对象，连接到Gmail：

```
>>> smtpObj = smtplib.SMTP('smtp.gmail.com', 587)

>>> type(smtpObj)

< class 'smtplib.SMTP'>
```

输入type(smtpObj)表明，smtpObj中保存了一个SMTP对象。你需要这个SMTP对象，以便调用它的方法，登录并发送电子邮件。如果

smtplib.SMTP()调用不成功，你的SMTP服务器可能不支持TLS端口587。在这种情况下，你需要利用smtplib.SMTP_SSL()和465端口，来创建SMTP对象。

```
>>> smtpObj = smtplib.SMTP_SSL('smtp.gmail.com', 465)
```

注意

如果没有连接到因特网，Python将抛出socket.gaierror: [Errno 11004] getaddrinfo failed或类似的异常。

对于你的程序，TLS和SSL之间的区别并不重要。只需要知道你的SMTP服务器使用哪种加密标准，这样就知道如何连接它。在接下来的所有交互式环境示例中，smtpObj变量将包含smtplib.SMTP()或smtplib.SMTP_SSL()函数返回的SMTP对象。

16.2.2 发送SMTP的“Hello”消息

得到SMTP对象后，调用它的名字古怪的EHLO()方法，向SMTP电子邮件服务器“打招呼”。这种问候是SMTP中的第一步，对于建立到服务器的连接是很重要的。你不需要知道这些协议的细节。只要确保得到SMTP对象后，第一件事就是调用ehlo()方法，否则以后的方法调用会导致错误。下面是一个ehlo()调用和返回值的例子：

```
>>> smtpObj.ehlo()

(250, b'mx.google.com at your service, [216.172.148.131]\nSIZE 35882577\
n8BITMIME\nSTARTTLS\nENHANCEDSTATUSCODES\nCHUNKING')
```

如果在返回的元组中，第一项是整数250（SMTP中“成功”的代码），则问候成功了。

16.2.3 开始TLS加密

如果要连接到SMTP服务器的587端口（即使用TLS加密），接下来需要调用starttls()方法。这是为连接实现加密必须的步骤。如果要连接到465端口（使用SSL），加密已经设置好了，你应该跳过这一步。

下面是starttls()方法调用的例子：

```
>>> smtpObj.starttls()
(220, b'2.0.0 Ready to start TLS')
```

starttls()让SMTP连接处于TLS模式。返回值220告诉你，该服务器已准备就绪。

16.2.4 登录到SMTP服务器

到SMTP服务器的加密连接建立后，可以调用login()方法，用你的用户名（通常是你的电子邮件地址）和电子邮件密码登录。

```
>>> smtpObj.login('my_email_address@gmail.com
', 'MY_SECRET_PASSWORD
```

```
')

(235, b'2.7.0 Accepted')
```

传入电子邮件地址字符串作为第一个参数，密码字符串作为第二个参数。返回值235表示认证成功。如果密码不正确，Python会抛出smtplib. SMTPAuthenticationError异常。

将密码放在源代码中要当心。如果有人复制了你的程序，他们就能访问你的电子邮件账户！调用input()，让用户输入密码是一个好主意。每次运行程序时输入密码可能不方便，但这种方法不会在未加密的文件中留下你的密码，黑客或笔记本电脑窃贼不会轻易地得到它。

16.2.5 发送电子邮件

登录到电子邮件提供商的SMTP服务器后，可以调用的sendmail()方法来发送电子邮件。sendmail()方法调用看起来像这样：

```
>>> smtpObj.sendmail('my_email_address@gmail.com

', 'recipient@example.com

',
```

```
'Subject: So long.\nDear Alice, so long and thanks for all the fish. Sincer

Bob')

{}
```

sendmail()方法需要三个参数。

- 你的电子邮件地址字符串（电子邮件的“from”地址）。
- 收件人的电子邮件地址字符串，或多个收件人的字符串列表（作为“to”地址）。
- 电子邮件正文字符串。

电子邮件正文字符串必须以'Subject: \n'开头，作为电子邮件的主题行。'\n'换行符将主题行与电子邮件的正文分开。

sendmail()的返回值是一个字典。对于电子邮件传送失败的每个收件人，该字典中会有一个键值对。空的字典意味着对所有收件人已成功发送电子邮件。

> **Gmail应用程序专用密码**
>
> Gmail有针对谷歌账户的附加安全功能，称为应用程序专用密码。如果当你的程序试图登录时，收到“需要应用程序专用密码”的错误信息，就必须在Python脚本设置这样一个密码。具体如何设置谷歌账户的应用程序专用密码，参见http://nostarch.com/automatestuff/ 。

16.2.6 从SMTP服务器断开

确保在完成发送电子邮件时，调用quit()方法。这让程序从SMTP服务器断开。

```
>>> smtpObj.quit()



(221, b'2.0.0 closing connection ko10sm23097611pbd.52 - gsmtp')
```

返回值221表示会话结束。

要复习连接和登录服务器、发送电子邮件和断开的所有步骤，请参阅 16.2节“发送电子邮件”。

16.3 IMAP

正如SMTP是用于发送电子邮件的协议，因特网消息访问协议（IMAP）规定了如何与电子邮件服务提供商的服务器通信，取回发送到你的电子邮件地址的电子邮件。Python带有一个imaplib模块，但实际上第三方的imapclient模块更易用。本章介绍了如何使用IMAPClient，完整的文档在http://imapclient.readthedocs.org/ 。

imapclient模块从IMAP服务器下载电子邮件，格式相当复杂。你很可能希望将它们从这种格式转换成简单的字符串。pyzmail模块替你完成解析这些邮件的辛苦工作。在http://www.magiksys.net/pyzmail/ 可以找到PyzMail的完整文档。

从终端窗口安装imapclient和pyzmail。附录A包含了如何安装第三方模块的步骤。

16.4 用IMAP获取和删除电子邮件

在Python中，查找和获取电子邮件是一个多步骤的过程，需要第三

方模块imapclient和pyzmail。作为概述，这里有一个完整的例子，包括登录到IMAP服务器，搜索电子邮件，获取它们，然后从中提取电子邮件的文本。

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
>>> imapObj.login('my_email_address@gmail.com
', 'MY_SECRET_PASSWORD
')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
>>> imapObj.select_folder('INBOX', readonly=True)
>>> UIDs = imapObj.search(['SINCE 05-Jul-2014'])
>>> UIDs
```

```
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
>>> rawMessages = imapObj.fetch([40041], ['BODY[]', 'FLAGS'])

>>> import pyzmail

>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])

>>> message.get_subject()

'Hello!'
>>> message.get_addresses('from')

[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')

[(Jane Doe', 'jdoe@example.com')]
>>> message.get_addresses('cc')
```

```
[]
>>> message.get_addresses('bcc')

[]
>>> message.text_part != None

True
>>> message.text_part.get_payload().decode(message.text_part.charset)

'Follow the money.\r\n\r\n-Ed\r\n'
>>> message.html_part != None

True
>>> message.html_part.get_payload().decode(message.html_part.charset)

'< div dir="ltr">< div>So long, and thanks for all the fish!< br>< br>< /di
Al< br>< /div>\r\n'
>>> imapObj.logout()
```

你不必记住这些步骤。在详细介绍每一步之后，你可以回来看这个概述，加强记忆。

16.4.1 连接到IMAP服务器

就像你需要一个SMTP对象连接到SMTP服务器并发送电子邮件一样，你需要一个IMAPClient对象，连接到IMAP服务器并接收电子邮件。首先，你需要电子邮件服务提供商的IMAP服务器域名。这和SMTP服务器的域名不同。表16-2列出了几个流行的电子邮件服务提供商的IMAP服务器。

表**16-2** 电子邮件提供商及其**IMAP**服务器

提供商	**IMAP**服务器域名
Gmail	*imap.gmail.com*
Outlook.com/Hotmail.com	*imap-mail.outlook.com*
Yahoo Mail	*imap.mail.yahoo.com*
AT&T	*imap.mail.att.net*
Comcast	*imap.comcast.net*
Verizon	*incoming.verizon.net*

得到IMAP服务器域名后，调用imapclient.IMAPClient()函数，创建一个IMAPClient对象。大多数电子邮件提供商要求SSL加密，传入SSL=TRUE关键字参数。在交互式环境中输入以下代码（使用你的提供商的域名）：

```
>>> import imapclient
>>> imapObj = imapclient.IMAPClient('imap.gmail.com', ssl=True)
```

在接下来的小节里所有交互式环境的例子中，imapObj变量将包含imapclient.IMAPClient()函数返回的IMAPClient对象。在这里，客户端是连接到服务器的对象。

16.4.2 登录到IMAP服务器

取得IMAPClient对象后，调用它的login()方法，传入用户名（这通常是你的电子邮件地址）和密码字符串。

```
>>> imapObj.login('my_email_address@gmail.com', 'MY_SECRET_PASSWORD')
'my_email_address@gmail.com Jane Doe authenticated (Success)'
```

要记住，永远不要直接在代码中写入密码！应该让程序从input()接受输入的密码。

如果IMAP服务器拒绝用户名/密码的组合，Python会抛出imaplib.error异常。对于Gmail账户，你可能需要使用应用程序专用的密码。详细信息请参阅16.2.5节中的“Gmail应用程序专用密码”。

16.4.3 搜索电子邮件

登录后，实际获取你感兴趣的电子邮件分为两步。首先，必须选择要搜索的文件夹。然后，必须调用IMAPClient对象的search()方法，传入IMAP搜索关键词字符串。

16.4.4 选择文件夹

几乎每个账户默认都有一个INBOX文件夹，但也可以调用IMAPClient对象的list_folders()方法，获取文件夹列表。这将返回一个元组的列表。每个元组包含一个文件夹的信息。输入以下代码，继续交互式环境的例子：

```
>>> import pprint
>>> pprint.pprint(imapObj.list_folders())
[(('\\HasNoChildren',), '/', 'Drafts'),
 (('\\HasNoChildren',), '/', 'Filler'),
 (('\\HasNoChildren',), '/', 'INBOX'),
 (('\\HasNoChildren',), '/', 'Sent'),
--snip
```

```
--
 (('\\HasNoChildren', '\\Flagged'), '/', '[Gmail]/Starred'),
 (('\\HasNoChildren', '\\Trash'), '/', '[Gmail]/Trash')]
```

如果你有一个Gmail账户，这就是输出可能的样子（Gmail将文件夹称为label，但它们的工作方式与文件夹相同）。每个元组的三个值，例如 (('\HasNoChildren',), '/', 'INBOX')，解释如下：

- 该文件夹的标志的元组（这些标志代表到底是什么超出了本书的讨论范围，你可以放心地忽略该字段）。
- 名称字符串中用于分隔父文件夹和子文件夹的分隔符。
- 该文件夹的全名。

要选择一个文件夹进行搜索，就调用IMAPClient对象的select_folder()方法，传入该文件夹的名称字符串。

```
>>> imapObj.select_folder('INBOX', readonly=True)
```

可以忽略select_folder()的返回值。如果所选文件夹不存在，Python会抛出imaplib.error异常。

readonly=True关键字参数可以防止你在随后的方法调用中，不小心更改或删除该文件夹中的任何电子邮件。除非你想删除的电子邮件，否则将readonly设置为True总是个好主意。

16.4.5 执行搜索

文件夹选中后，就可以用IMAPClient对象的search()方法搜索电子邮件。search()的参数是一个字符串列表，每一个格式化为IMAP搜索键。表16-3介绍了各种搜索键。

表**16-3 IMAP**搜索键

搜索键	含义
'ALL'	返回该文件夹中的所有邮件。如果你请求一个大文件夹中的所有消息，可能会遇到imaplib的大小限制。参见16.4.6小节“大小限制”
'BEFORE *date* ', 'ON *date* ', 'SINCE *date* '	这三个搜索键分别返回给定date之前、当天和之后IMAP服务器接收的消息。日期的格式必须像05-Jul-2015。此外，虽然'SINCE 05-Jul-2015'将匹配7月5日当天和之后的消息，但'BEFORE 05-Jul-2015'仅匹配7月5日之前的消息，不包括7月5日当天
'SUBJECT *string* ', 'BODY *string* ', 'TEXT *string* '	分别返回string出现在主题、正文、主题或正文中的消息。如果string中有空格，就使用双引号：'TEXT "search with spaces"'
'FROM *string* ', 'TO *string* ', 'CC *string* ', 'BCC *string* '	返回所有消息，其中string分别出现在“from”邮件地址，“to”邮件地址，“cc”（抄送）地址，或“bcc”（密件抄送）地址中。 如果string中有多个电子邮件地址，就用空格将它们分开，并使用双引号： 'CC "*firstcc@example.com secondcc@example.com* "'
'SEEN', 'UNSEEN'	分别返回包含和不包含\ Seen标记的所有信息。如果电子邮件已经被fetch()方法调用访问（稍后描述），或者你曾在电子邮件程序或网络浏览器中点击过它，就会有\ Seen标记。比较常用的说法是电子邮件“已读”，而不是“已看”，但它们的意思一样。
'ANSWERED', 'UNANSWERED'	分别返回包含和不包含\ Answered标记的所有消息。如果消息已答复，就会有\ Answered标记
'DELETED', 'UNDELETED'	分别返回包含和不包含*Deleted* 标记的所有信息。用delete_messages()方法删除的邮件就会有*Deleted* 标记，直到调用expunge()方法才会永久删除（请参阅16.4.10节“删除电子邮

	件”）。请注意，一些电子邮件提供商，例如Gmail，会自动清除邮件
'DRAFT', 'UNDRAFT'	分别返回包含和不包含\ Draft标记的所有消息。草稿邮件通常保存在单独的草稿文件夹中，而不是在收件箱中
'FLAGGED', 'UNFLAGGED'	分别返回包含和不包含*Flagged* 标记的所有消息。这个标记通常用来标记电子邮件为“重要”或“紧急”
'LARGER *N* ', 'SMALLER *N* '	分别返回大于或小于N个字节的所有消息
'NOT *search-key* '	返回搜索键不会返回的那些消息
'OR *search-key1 search-key2* '	返回符合第一个或第二个搜索键的消息

请注意，在处理标志和搜索键方面，某些IMAP服务器的实现可能稍有不同。可能需要在交互式环境中试验一下，看看它们实际的行为如何。

在传入search()方法的列表参数中，可以有多个IMAP搜索键字符串。返回的消息将匹配所有的搜索键。如果想匹配任何一个搜索键，使用OR搜索键。对于NOT和OR搜索键，它们后边分别跟着一个和两个完整的搜索键。

下面是search()方法调用的一些例子，以及它们的含义：

imapObj.search (['ALL']) 返回当前选定的文件夹中的每一个消息。

imapObj.search (['ON 05-Jul-2015'])返回在2015年7月5日发送的每个消息。

imapObj.search(['SINCE 01-Jan-2015', 'BEFORE 01-Feb-2015', 'UNSEEN']) 返回2015年1月发送的所有未读消息（注意，这意味着从1

月1日直到2月1日，但不包括2月1日）。

imapObj.search(['SINCE 01-Jan-2015', 'FROM alice@example.com']) 返回自2015年开始以来，发自alice@example.com的消息。

imapObj.search(['SINCE 01-Jan-2015', 'NOT FROM alice@example.com']) 返回自2015年开始以来，除alice@example.com外，其他所有人发来的消息。

imapObj.search(['OR FROM alice@example.com FROM bob@example.com']) 返回发自alice@example.com或bob@example.com的所有信息。

imapObj.search(['FROM alice@example.com', 'FROM bob@example.com'])恶作剧 例子！该搜索不会返回任何消息，因为消息必须匹配所有搜索关键词。因为只能有一个“from”地址，所以一条消息不可能既来自alice@example.com，又来自bob@example.com。

search()方法不返回电子邮件本身，而是返回邮件的唯一整数ID（UID）。然后，可以将这些UID传入fetch()方法，获得邮件内容。

输入以下代码，继续交互式环境的例子：

```
>>> UIDs = imapObj.search(['SINCE 05-Jul-2015'])
>>> UIDs
[40032, 40033, 40034, 40035, 40036, 40037, 40038, 40039, 40040, 40041]
```

这里，search()返回的消息ID列表（针对7月5日以来接收的消息）保存在UIDs中。计算机上返回的UIDs列表与这里显示的不同，它们对于特定的电子邮件账户是唯一的。如果你稍后将UID传递给其他函数调用，请用你收到的UID值，而不是本书例子中打印的。

16.4.6 大小限制

如果你的搜索匹配大量的电子邮件，Python可能抛出异常imaplib.error: got more than 10000 bytes。如果发生这种情况，必须断开并重连IMAP服务器，然后再试。

这个限制是防止Python程序消耗太多内存。遗憾的是，默认大小限制往往太小。可以执行下面的代码，将限制从10000字节改为10000000字节：

```
>>> import imaplib
>>> imaplib._MAXLINE = 10000000
```

这应该能避免该错误消息再次出现。也许要在你写的每一个IMAP程序中加上这两行。

16.4.7 取邮件并标记为已读

得到UID的列表后，可以调用IMAPClient对象的fetch()方法，获得实际的电子邮件内容。

UID列表是fetch()的第一个参数。第二个参数应该是['BODY[]']，它

告诉fetch()下载UID列表中指定电子邮件的所有正文内容。

使用IMAPClient的gmail_search()方法

如果登录到imap.gmail.com服务器来访问Gmail账户，IMAPClient对象提供了一个额外的搜索函数，模拟Gmail网页顶部的搜索栏，如图16-1中高亮的部分所示。

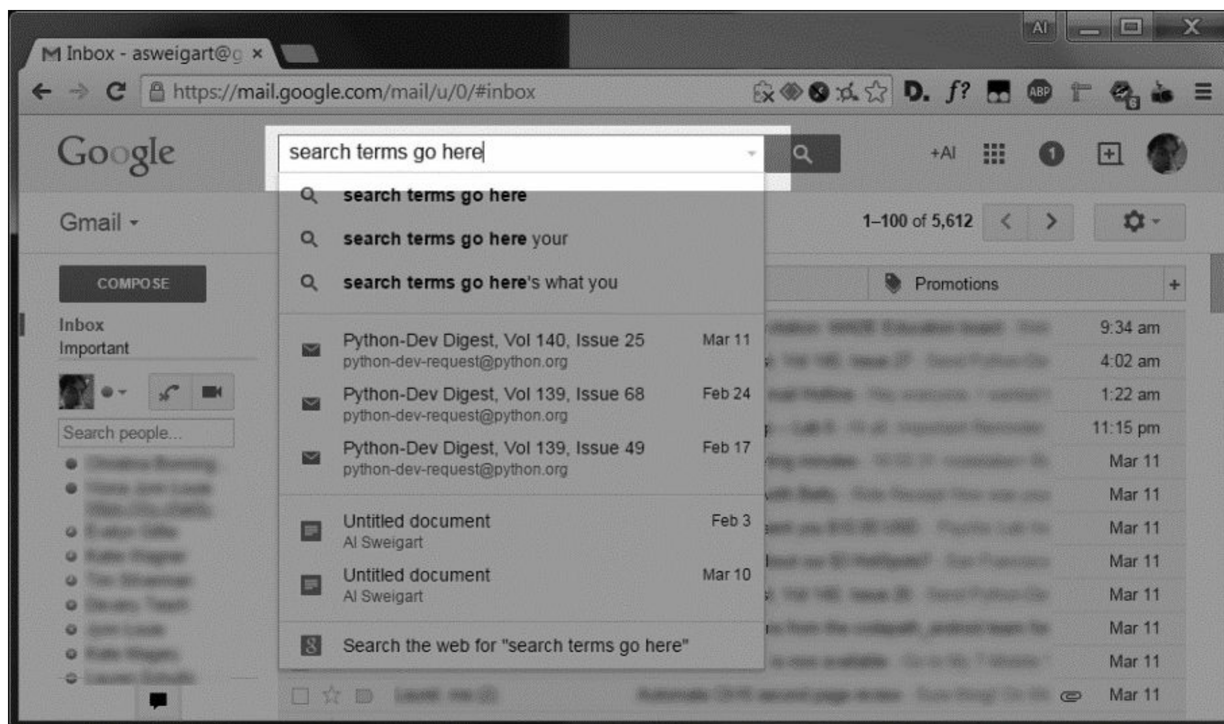


图16-1 在Gmail网页顶部的搜索栏

除了用IMAP搜索键搜索，可以使用Gmail更先进的搜索引擎。Gmail在匹配密切相关的单词方面做得很好（例如，搜索driving也会匹配drive和drove），并按照匹配的程度对搜索结果排序。也可以使用Gmail的高级搜索操作符（更多信息请参见http://nostarch.com/automatestuff/）。如果登录到Gmail账户，向gmail_search()方法传入搜索条件，而不是search()方法，就像下面交互式环境的例子：

```
>>> UIDs = imapObj.gmail_search('meaning of life')

>> UIDs

[42]
```

> 啊，是的，那封电子邮件包含了生命的意义！我一直在期待。

让我们继续交互式环境的例子。

```
>>> rawMessages = imapObj.fetch(UIDs, ['BODY[]'])
>>> import pprint
>>> pprint.pprint(rawMessages)
{40040: {'BODY[]': 'Delivered-To: my_email_address@gmail.com\r\n'
                   'Received: by 10.76.71.167 with SMTP id '
--snip
--
                   '\r\n'
                   '------=_Part_6000970_707736290.1404819487066--\r\n',
         'SEQ': 5430}}
```

导入 pprint，将 fetch()的返回值（保存在变量 rawMessages 中）传入pprint.pprint()，“漂亮打印”它。你会看到，这个返回值是消息的嵌套字典，其中以UID作为键。每条消息都保存为一个字典，包含两个键：'BODY[]'和'SEQ'。'BODY[]'键映射到电子邮件的实际正文。'SEQ'键是序列号，它与UID的作用类似。你可以放心地忽略它。

正如你所看到的，在'BODY[]'键中的消息内容是相当难理解的。这种格式称为RFC822，是专为IMAP服务器读取而设计的。但你并不需要理解RFC 822格式，本章稍后的pyzmail模块将替你来理解它。

如果你选择一个文件夹进行搜索，就用readonly=True关键字参数来调用select_ folder()。这样做可以防止意外删除电子邮件，但这也意味着你用fetch()方法获取邮件时，它们不会标记为已读。如果确实希望在获取邮件时将它们标记已读，就需要将readonly=False传入select_folder()。如果所选文件夹已处于只读模式，可以用另一个 select_folder()调用重新选择当前文件夹，这次用readonly=False关键字参数：

```
>>> imapObj.select_folder('INBOX', readonly=False)
```

16.4.8 从原始消息中获取电子邮件地址

对于只想读邮件的人来说，fetch()方法返回的原始消息仍然不太有用。pyzmail模块解析这些原始消息，将它们作为PyzMessage对象返回，使邮件的主题、正文、“收件人”字段、“发件人”字段和其他部分能用Python代码轻松访问。

用下面的代码继续交互式环境的例子（使用你自己的邮件账户的UID，而不是这里显示的）：

```
>>> import pyzmail
>>> message = pyzmail.PyzMessage.factory(rawMessages[40041]['BODY[]'])
```

首先，导入pyzmail。然后，为了创建一个电子邮件的PyzMessage对象，调用pyzmail.PeekMessage.factory()函数，并传入原始邮件的'BODY[]'部分。结果保存在message中。现在，message中包含一个PyzMessage对象，它有几个方法，可以很容易地获得的电子邮件主题行，以及所有发件人和收件人的地址。get_subject()方法将主题返回为一个简单字符串。get_addresses()方法针对传入的字段，返回一个地址列表。例如，该方法调用可能像这样：

```
>>> message.get_subject()
'Hello!'
>>> message.get_addresses('from')
[('Edward Snowden', 'esnowden@nsa.gov')]
>>> message.get_addresses('to')
[(Jane Doe', 'my_email_address@gmail.com')]
>>> message.get_addresses('cc')
[]
>>> message.get_addresses('bcc')
```

```
[]
```

请注意，get_addresses()的参数是'from'、'to'、'cc'或 'bcc'。get_addresses()的返回值是一个元组列表。每个元组包含两个字符串：第一个是与该电子邮件地址关联的名称，第二个是电子邮件地址本身。如果请求的字段中没有地址，get_addresses()返回一个空列表。在这里，'cc'抄送和'bcc'密件抄送字段都没有包含地址，所以返回空列表。

16.4.9 从原始消息中获取正文

电子邮件可以是纯文本、HTML 或两者的混合。纯文本电子邮件只包含文本，而HTML电子邮件可以有颜色、字体、图像和其他功能，使得电子邮件看起来像一个小网页。如果电子邮件仅仅是纯文本，它的PyzMessage对象会将html_part属性设为None。同样，如果电子邮件只是HTML，它的PyzMessage对象会将text_part属性设为None。

否则，text_part或html_part将有一个get_payload()方法，将电子邮件的正文返回为bytes数据类型（bytes数据类型超出了本书的范围）。但是，这仍然不是我们可以使用的字符串。啊！最后一步对get_payload()返回的bytes值调用decode()方法。decode()方法接受一个参数：这条消息的字符编码，保存在text_part.charset或html_part.charset属性中。最后，这返回了邮件正文的字符串。

输入以下代码，继续交互式环境的例子：

```
❶ >>> message.text_part != None
  True
```

```
  >>> message.text_part.get_payload().decode(message.text_part.charset)
❷ 'So long, and thanks for all the fish!\r\n\r\n-Al\r\n'
❸ >>> message.html_part != None
  True
❹ >>> message.html_part.get_payload().decode(message.html_part.charset)
  '< div dir="ltr">< div>So long, and thanks for all the fish!< br>< br>< /
  < br>< /div>\r\n'
```

我们正在处理的电子邮件包含纯文本和HTML内容，因此保存在message中的PyzMessage对象的text_part和html_part属性不等于None❶❸。对消息的text_part调用get_payload()，然后在bytes值上调用decode()，返回电子邮件的文本版本的字符串❷。对消息的html_part调用get_payload()和decode()，返回电子邮件的HTML版本的字符串❹。

16.4.10 删除电子邮件

要删除电子邮件，就向IMAPClient对象的delete_messages()方法传入一个消息UID的列表。这为电子邮件加上\Deleted标志。调用expunge()方法，将永久删除当前选中的文件夹中带\Deleted标志的所有电子邮件。请看下面的交互式环境的例子：

```
❶ >>> imapObj.select_folder('INBOX', readonly=False)
```

```
❷ >>> UIDs = imapObj.search(['ON 09-Jul-2015'])
   >>> UIDs
   [40066]
   >>> imapObj.delete_messages(UIDs)
❸ {40066: ('\\Seen', '\\Deleted')}
   >>> imapObj.expunge()
   ('Success', [(5452, 'EXISTS')])
```

这里，我们调用了IMAPClient对象的select_folder()方法，传入'INBOX'作为第一个参数，选择了收件箱。我们也传入了关键字参数readonly=False，这样我们就可以删除电子邮件❶。我们搜索收件箱中的特定日期收到的消息，将返回的消息ID保存在UIDs中❷。调用delete_message()并传入UIDs，返回一个字典，其中每个键值对是一个消息 ID 和消息标志的元组，它现在应该包含\Deleted标志❸。然后调用expunge()，永久删除带\Deleted标志的邮件。如果清除邮件没有问题，就返回一条成功信息。请注意，一些电子邮件提供商，如Gmail，会自动清除用delete_messages()删除的电子邮件，而不是等待来自IMAP客户

端的expunge命令。

16.4.11 从IMAP服务器断开

如果程序已经完成了获取和删除电子邮件，就调用IMAPClient的logout()方法，从IMAP服务器断开连接。

```
>>> imapObj.logout()
```

如果程序运行了几分钟或更长时间，IMAP服务器可能会超时，或自动断开。在这种情况下，接下来程序对IMAPClient对象的方法调用会抛出异常，像下面这样：

```
imaplib.abort: socket error: [WinError 10054] An existing connection was
forcibly closed by the remote host
```

在这种情况下，程序必须调用imapclient.IMAPClient()，再次连接。

哟！齐活了。要跳过很多圈圈，但你现在有办法让Python程序登录到一个电子邮件账户，并获取电子邮件。需要回忆所有步骤时，你可以随时参考16.4节“用IMAP获取和删除电子邮件”。

16.5 项目：向会员发送会费提醒电子邮件

假定你一直“自愿”为“强制自愿俱乐部”记录会员会费。这确实是一项枯燥的工作，包括维护一个电子表格，记录每个月谁交了会费，并用电子邮件提醒那些没交的会员。不必你自己查看电子表格，而是向会费

超期的会员复制和粘贴相同的电子邮件。你猜对了，让我们编写一个脚本，帮你完成任务。

在较高的层面上，下面是程序要做的事：

- 从Excel电子表格中读取数据。
- 找出上个月没有交费的所有会员。
- 找到他们的电子邮件地址，向他们发送针对个人的提醒。

这意味着代码需要做到以下几点：

- 用openpyxl模块打开并读取Excel文档的单元格（处理Excel文件参见第12章）。
- 创建一个字典，包含会费超期的会员。
- 调用smtplib.SMTP()、ehlo()、starttls()和login()，登录SMTP服务器。
- 针对会费超期的所有会员，调用sendmail()方法，发送针对个人的电子邮件提醒。

打开一个新的文件编辑器窗口，并保存为sendDuesReminders.py。

第1步：打开Excel文件

假定用来记录会费支付的 Excel 电子表格看起来如图 16-2 所示，放在名为duesRecords.xlsx的文件中。可以从http://nostarch.com/automatestuff/下载该文件。

Member	Email	Jan 2014	Feb 2014	Mar 2014	Apr 2014	May 2014	Jun 2014
Alice	alice@example.com	paid	paid	paid	paid	paid	
Bob	bob@example.com	paid	paid	paid	paid		
Carol	carol@example.com	paid	paid	paid	paid	paid	paid
David	david@example.com	paid	paid	paid	paid	paid	paid
Eve	eve@example.com	paid	paid	paid			
Fred	fred@example.com	paid	paid	paid	paid	paid	paid

图16-2 记录会员会费支付电子表格

该电子表格中包含每个成员的姓名和电子邮件地址。每个月有一列，记录会员的付款状态。在成员交纳会费后，对应的单元格就记为paid。

该程序必须打开duesRecords.xlsx，通过调用get_highest_column()方法，弄清楚最近一个月的列（可以参考第12章，了解用openpyxl模块访问Excel电子表格文件单元格的更多信息）。在文件编辑器窗口中输入以下代码：

```
  #! python3
  # sendDuesReminders.py - Sends emails based on payment status in spreadsh

  import openpyxl, smtplib, sys

  # Open the spreadsheet and get the latest dues status.
❶ wb = openpyxl.load_workbook('duesRecords.xlsx')
❷ sheet = wb.get_sheet_by_name('Sheet1')

❸ lastCol = sheet.get_highest_column()
❹ latestMonth = sheet.cell(row=1, column=lastCol).value

  # TODO: Check each member's payment status.
```

```
# TODO: Log in to email account.

# TODO: Send out reminder emails.
```

导入openpyxl、smtplib和sys模块后，我们打开duesRecords.xlsx文件，将得到的Workbook对象保存在wb中❶。然后，取得Sheet 1，将得到的Worksheet对象保存在sheet中❷。既然有了Worksheet对象，就可以访问行、列和单元格。我们将最后一列保存在lastCol中❸，然后用行号1和lastCol来访问应该记录着最近月份的单元格。取得该单元格的值，并保存在latestMonth 中❹。

第2步：查找所有未付成员

一旦确定了最近一个月的列数（保存在lastCol中），就可以循环遍历第一行（这是列标题）之后的所有行，看看哪些成员在该月会费的单元格中写着paid。如果会员没有支付，就可以从列1和2中分别抓取成员的姓名和电子邮件地址。这些信息将放入unpaidMembers字典，它记录最近一个月没有交费的所有成员。将以下代码添加到sendDuesReminder.py中。

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadsh

--snip

--

# Check each member's payment status.

unpaidMembers = {}
```

```
❶ for r in range(2, sheet.get_highest_row() + 1):
❷      payment = sheet.cell(row=r, column=lastCol).value
       if payment != 'paid':
❸           name = sheet.cell(row=r, column=1).value
❹           email = sheet.cell(row=r, column=2).value
❺           unpaidMembers[name] = email
```

这段代码设置了一个空字典unpaidMembers，然后循环遍历第一行之后所有的行❶。对于每一行，最近月份的值保存在payment中❷。如

果payment不等于'paid'，则第一列的值保存在name中❸，第二列的值保存在email中❹，name和email添加到unpaidMembers中❺。

第3步：发送定制的电子邮件提醒

得到所有未付费成员的名单后，就可以向他们发送电子邮件提醒了。将下面的代码添加到程序中，但要代入你的真实电子邮件地址和提供商的信息：

```
#! python3
# sendDuesReminders.py - Sends emails based on payment status in spreadshee

--snip

--

# Log in to email account.

smtpObj = smtplib.SMTP('smtp.gmail.com', 587)

smtpObj.ehlo()

smtpObj.starttls()

smtpObj.login('my_email_address@gmail.com
```

```
', sys.argv[1])
```

调用smtplib.SMTP()并传入提供商的域名和端口，创建一个SMTP对象。调用ehlo()和starttls()，然后调用login()，并传入你的电子邮件地址和sys.argv[1]，其中保存着你的密码字符串。在每次运行程序时，将密码作为命令行参数输入，避免在源代码中保存密码。

程序登录到你的电子邮件账户后，就应该遍历unpaidMembers字典，向每个会员的电子邮件地址发送针对个人的电子邮件。将以下代码添加到sendDuesReminders.py：

```
  #! python3
  # sendDuesReminders.py - Sends emails based on payment status in spreadsh

  --snip

--

  # Send out reminder emails.

  for name, email in unpaidMembers.items():
```

```
❶      body = "Subject: %s dues unpaid.\nDear %s,\nRecords show that you ha
  paid dues for %s. Please make this payment as soon as possible. Thank you
  (latestMonth, name, latestMonth)
❷      print('Sending email to %s...' % email)
❸      sendmailStatus = smtpObj.sendmail('my_email_address@gmail.com
', email, body)

❹      if sendmailStatus != {}:
           print('There was a problem sending email to %s: %s' % (email,
```

```
        sendmailStatus))



  smtpObj.quit()
```

这段代码循环遍历unpaidMembers中的姓名和电子邮件。对于每个没有付费的成员，我们用最新的月份和成员的名称，定制了一条消息，并保存在body中❶。我们打印输出，表示正在向这个会员的电子邮件地址发送电子邮件❷。然后调用sendmail()，向它传入地址和定制的消息❸。返回值保存在sendmailStatus中。

回忆一下，如果SMTP服务器在发送某个电子邮件时报告错误，sendmail()方法将返回一个非空的字典值。for循环最后部分在❹行检查返回的字典是否非空，如果非空，则打印收件人的电子邮件地址以及返回的字典。

程序完成发送所有电子邮件后，调用quit()方法，与SMTP服务器断开连接。

如果运行该程序，输出会像这样：

```
Sending email to alice@example.com...
Sending email to bob@example.com...
Sending email to eve@example.com...
```

收件人将收到如图16-3所示的电子邮件。

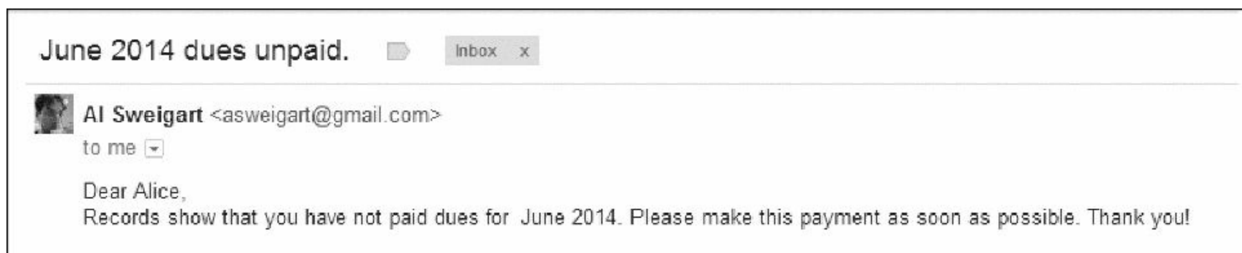


图16-3 从sendDuesReminders.py自动发送的电子邮件

16.6 用Twilio发送短信

大多数人更可能靠近自己的手机，而不是自己的电脑，所以与电子邮件相比，短信发送通知可能更直接、可靠。此外，短信的长度较短，让人更有可能阅读它们。

在本节中，你将学习如何注册免费的Twilio服务，并用它的Python模块发送短信。Twilio是一个SMS网关服务，这意味着它是一种服务，让你通过程序发送短信。虽然每月发送多少短信会有限制，并且文本前面会加上Sent from a Twilio trial account，但这项试用服务也许能满足你的个人程序。免费试用没有限期，不必以后升级到付费的套餐。

Twilio不是唯一的SMS网关服务。如果你不喜欢使用Twilio，可以在线搜索free sms gateway、python sms api，甚至twilio alternatives，寻找替代服务。

注册Twilio账户之前，先安装twilio模块。附录A详细介绍了如何安装第三方模块。

本节特别针对美国。Twilio 确实也在美国以外的国家提供手机短信服务，本书并不包括这些细节。但twilio 模块及其功能，在美国以外的国家也能用。更多信息请参见http://twilio.com/。

16.6.1 注册Twilio账号

访问http://twilio.com/并填写注册表单。注册了新账户后，你需要验证一个手机号码，短信将发给该号码（这项验证是必要的，防止有人利

用该服务向任意的手机号码发送垃圾短信）。

收到验证号码短信后，在Twilio网站上输入它，证明你拥有要验证的手机。现在，就可以用twilio模块向这个电话号码发送短信了。

Twilio提供的试用账户包括一个电话号码，它将作为短信的发送者。你将需要两个信息：你的账户SID和AUTH（认证）标志。在登录Twilio账户时，可以在Dashboard页面上找到这些信息。从Python程序登录时，这些值将作为你的Twilio用户名和密码。

16.6.2 发送短信

一旦安装了twilio模块，注册了Twilio账号，验证了你的手机号码，登记了Twilio电话号码，获得了账户的SID和auth标志，你就终于准备好通过Python脚本向你自己发短信了。

与所有的注册步骤相比，实际的Python代码很简单。保持计算机连接到因特网，在交互式环境中输入以下代码，用你的真实信息替换accountSID、authToken、myTwilioNumber和myCellPhone变量的值：

```
❶ >>> from twilio.rest import TwilioRestClient



  >>> accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx


'



  >>> authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
'

❷ >>> twilioCli = TwilioRestClient(accountSID, authToken)

  >>> myTwilioNumber = '+14955551234'

  >>> myCellPhone = '+14955558888'

❸ >>> message = twilioCli.messages.create(body='Mr. Watson - Come here - I

  to see you.', from_=myTwilioNumber, to=myCellPhone)
```

键入最后一行后不久，你会收到一条短信，内容为：Sent from your Twilio trial account - Mr. Watson - Come here – I want to see you.。

因为twilio模块的设计方式，导入它时需要使用from twilio.rest import TwilioRestClient，而不仅仅是import twilio❶。将账户的SID保存

在accountSID，认证标志保存在authToken中，然后调用TwilioRestClient()，并传入accountSID和authToken。TwilioRestClient()调用返回一个TwilioRestClient对象❷。该对象有一个message属性，该属性又有一个create()方法，可以用来发送短信。正是这个方法，将告诉Twilio的服务器发送短信。将你的Twilio号码和手机号码分别保存在myTwilioNumber和myCellPhone中，然后调用create()，传入关键字参数，指明短信的正文、发件人的号码（myTwilioNumber），以及收信人的电话号码（myCellPhone）❸

create()方法返回的Message对象将包含已发送短信的相关信息。输入以下代码，继续交互式环境的例子：

```
>>> message.to
'+14955558888'
>>> message.from
_
'+14955551234'
>>> message.body
'Mr. Watson - Come here - I want to see you.'
```

to、from和*body*属性应该分别保存了你的手机号码、*Twilio*号码和消息。请注意，发送手机号码是在*from* 属性中，末尾有一个下划线，而不是from。这是因为from是一个Python关键字（例如，你在from modulename import *形式的import语句中见过它），所以它不能作为一

个属性名。输入以下代码，继续交互式环境的例子：

```
>>> message.status
'queued'
>>> message.date_created
datetime.datetime(2015, 7, 8, 1, 36, 18)
>>> message.date_sent == None
True
```

status 属性应该包含一个字符串。如果消息被创建和发送，date_created 和date_sent属性应该包含一个datetime对象。如果已收到短信，而status属性却设置为'queued'，date_sent属性设置为None，这似乎有点奇怪。这是因为你先将Message对象记录在message变量中，然后短信才实际发送。你需要重新获取Message对象，查看它最新的status和date_sent。每个Twilio消息都有唯一的字符串ID（SID），可用于获取Message对象的最新更新。输入以下代码，继续交互式环境的例子：

```
  >>> message.sid
  'SM09520de7639ba3af137c6fcb7c5f4b51'
❶ >>> updatedMessage = twilioCli.messages.get(message.sid)
```

```
>>> updatedMessage.status
'delivered'
>>> updatedMessage.date_sent
datetime.datetime(2015, 7, 8, 1, 36, 18)
```

输入message.sid将显示这个消息的SID。将这个SID传入Twilio客户端的get()方法❶，你可以取得一个新的Message对象，包含最新的信息。在这个新的Message对象中，status和date_sent属性是正确的。

status属性将设置为下列字符串之一：'queued'、'sending'、'sent'、'delivered'、'undelivered'或'failed'。这些状态不言自明，但对于更准确的细节，请查看http://nostarch.com/automatestuff/的资源。

> **用Python接收短信**
>
> 遗憾的是，用Twilio接收短信比发送短信更复杂一些。Twilio需要你有一个网站，运行自己的Web应用程序。这已超出了本书的范围，但你可以在本书的资源中找到更多细节（http://nostarch.com/automatestuff/）。

16.7 项目：“只给我发短信”模块

最常用你的程序发短信的人可能就是你。当你远离计算机时，短信是通知你自己的好方式。如果你已经用程序自动化了一个无聊的任务，

它需要运行几小时，你可以在它完成时，让它用短信通知你。或者可以定期运行某个程序，它有时需要与你联系，例如天气检查程序，用短信提醒你带伞。

举一个简单的例子，下面是一个Python小程序，包含了textmyself()函数，它将传入的字符串参数作为短信发出。打开一个新的文件编辑器窗口，输入以下代码，用自己的信息替换帐户SID，认证标志和电话号码。将它保存为textMyself.py。

```
  #! python3
  # textMyself.py - Defines the textmyself() function that texts a message
  # passed to it as a string.

  # Preset values:
  accountSID = 'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
'
  authToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
'
  myNumber = '+15559998888'
  twilioNumber = '+15552225678'

  from twilio.rest import TwilioRestClient

❶ def textmyself(message):
❷      twilioCli = TwilioRestClient(accountSID, authToken)
❸      twilioCli.messages.create(body=message, from_=twilioNumber, to=myNum
```

该程序保存了账户的SID、认证标志、发送号码及接收号码。然后它定义了textmyself()，接收参数❶，创建TwilioRestClient对象❷，并用你传入的消息调用create()❸。

如果你想让其他程序使用textmyself()函数，只需将textMyself.py文件和Python的可执行文件放在同一个文件夹中（Windows上是C:\Python34，OS X上是/usr/local/lib/python3.4，Linux上是/usr/bin/python3）。现在，你可以在其他程序中使用该函数。只要想在程序中发短信给你，就添加以下代码：

```
import textmyself
textmyself.textmyself('The boring task is finished.')
```

注册Twilio和编写短信代码只要做一次。在此之后，从任何其他程序中发短信，只要两行代码。

16.8 小结

通过因特网和手机网络，我们用几十种不同的方式相互通信，但以电子邮件和短信为主。你的程序可以通过这些渠道沟通，这给它们带来强大的新通知功能。甚至可以编程运行在不同的计算机上，相互直接通过电子邮件能信，一个程序用SMTP发送电子邮件，另一个用IMAP收取。

Python 的 smtplib 提供了一些函数，利用 SMTP，通过电子邮件提供商的SMTP服务器发送电子邮件。同样，第三方的imapclient和pyzmail模块让你访问IMAP服务器，并取回发送给你的电子邮件。虽然IMAP比SMTP复杂一些，但它也相当强大，允许你搜索特定电子邮件、下载它们、解析它们，提取主题和正文作为字符串值。

短信与电子邮件有点不同，因为它不像电子邮件，发送短信不仅需要互联网连接。好在，像Twilio这样的服务提供了模块，允许你通过程序发送短信。一旦通过了初始设置过程，就能够只用几行代码来发送短信。掌握了这些模块，就可以针对特定的情况编程，在这些情况下发送通知或提醒。现在，你的程序将超越运行它们的计算机！

16.9 习题

1．发送电子邮件的协议是什么？检查和接收电子邮件的协议是什么？

2．必须调用哪4个smtplib函数/方法，才能登录到SMTP服务器？

3．必须调用哪两个imapclient函数/方法，才能登录到IMAP服务器？

4．传递给mapObj.search()什么样的参数？

5．如果你的代码收到了错误消息，got more than 10000 bytes，你该怎么做？

6．imapclient模块负责连接到IMAP服务器和查找电子邮件。什么模块负责读取imapclient收集的电子邮件？

7．在发送短信之前，你需要从Twilio得到哪3种信息？

16.10 实践项目

作为实践，编程完成以下任务。

16.10.1 随机分配家务活的电子邮件程序

编写一个程序，接受一个电子邮件地址的列表，以及一个需要做的家务活列表，并随机将家务活分配给他们。用电子邮件通知每个人分配给他们的家务。如果你觉得需要挑战，就记录每个人之前分配家务活的记录，这样就可以确保程序不会向任何人分配上一次同样的家务活。另一个可能的功能，就是安排程序每周自动运行一次。

这里有一个提示：如果将一个列表传入random.choice()函数，它将从该列表中返回一个随机选择的项。你的部分代码看起来可能像这样：

```
chores = ['dishes', 'bathroom', 'vacuum', 'walk dog']
randomChore = random.choice(chores)
chores.remove(randomChore)     # this chore is now taken, so remove it
```

16.10.2 伞提醒程序

第11章展示了如何利用requests模块，从http://weather.gov/ 抓取数据。编写一个程序，在你早晨快醒来时运行，检查当天是否会下雨。如果会下雨，让程序用短信提醒你出门之前带好一把伞。

16.10.3 自动退订

编程扫描你的电子邮件账户，在所有邮件中找到所有退订链接，并自动在浏览器中打开它们。该程序必须登录到你的电子邮件服务提供商的IMAP服务器，并下载所有电子邮件。可以用BeautifulSoup（在第11章中介绍）检查所有出现unsubscribe（退订）的HTML链接标签。

得到这些URL的列表后，可以用webbrowser.open()，在浏览器中自动打开所有这些链接。

仍然需要手工操作并完成所有额外的步骤，从这些邮件列表中退订。在大多数情况下，这需要点击一个链接确认。

但这个脚本让你不必查看所有电子邮件，寻找退订链接。然后，可以将这个脚本转给你的朋友，让他们能够针对他们的电子邮件账户运行它（要确保你的邮箱密码没有硬编码在源代码中）。

16.10.4 通过电子邮件控制你的电脑

编写一个程序，每15分钟检查电子邮件账户，获取用电子邮件发送的所有指令，并自动执行这些指令。例如，BitTorrent是一个对等网络下载系统。利用免费的BitTorrent软件，如qBittorrent，可以在家用电脑上下载很大的媒体文件。如果你用电子邮件向该程序发送一个（完全合法的，根本不是盗版的）BitTorrent链接，该程序将检查电子邮件，发现这个消息，提取链接，然后启动qBittorrent，开始下载文件。通过这种方式，你可以在离开家的时候让家用电脑开始下载，这些（完全合法的，根本不是盗版的）下载在你回家前就能完成。

第15章介绍了如何利用subprocess.Popen()函数启动计算机上的程序。例如，下面的调用将启动qBittorrent程序，并打开一个torrent文件：

```
qbProcess = subprocess.Popen(['C:\\Program Files (x86)\\qBittorrent\\
qbittorrent.exe', 'shakespeare_complete_works.torrent'])
```

当然，你希望该程序确保邮件来自于你自己。具体来说，你可能希望该邮件包含一个密码，因为在电子邮件中伪造“from”地址，对黑客来说很容易。该程序应该删除它发现的邮件，这样就不会每次检查电子邮件账户时重复执行命令。作为一个额外的功能，让程序每次执行命令时，用电子邮件或短信给你发一条确认信息。因为该程序运行时，你不会坐在运行它的计算机前面，所以利用日志函数（参见第 10章）写文本文件日志是一个好主意，你可以检查是否发生错误。

qBittorrent（以及其他BitTorrent应用程序）有一个功能，下载完成后，它可以自动退出。第15章解释了如何用Popen对象的wait()方法，确定启动的应用程序何时已经退出。wait()方法调用将阻塞，直到qBittorrent停止，然后程序可以通过电子邮件或短信，通知你下载已经完成。

可以为这个项目添加许多可能的功能。如果遇到困难，可以从http://nostarch. com/automatestuff/下载这个程序的示例实现。

第17章　操作图像

如果你有一台数码相机，或者只是将照片从手机上传到Facebook，你可能随时都会偶然遇到数字图像文件。你可能知道如何使用基本的图形软件，如Microsoft Paint或Paintbrush，甚至更高级的应用程序，如Adobe Photoshop。但是，如果需要编辑大量的图像，手工编辑可能是漫长、枯燥的工作。

请用Python。Pillow是一个第三方Python模块，用于处理图像文件。该模块包含一些函数，可以很容易地裁剪图像、调整图像大小，以及编辑图像的内容。可以像Microsoft Paint或Adobe Photoshop 一样处理图像，有了这种能力，Python 可以轻松地自动编辑成千上万的图像。

17.1 计算机图像基础

为了处理图像，你需要了解计算机如何处理图像中的颜色和坐标的基本知识，以及如何在Pillow中处理颜色和坐标。但在继续探讨之前，先要安装pillow模块。安装第三方模块请见附录A。

17.1.1 颜色和RGBA值

计算机程序通常将图像中的颜色表示为RGBA值。RGBA值是一组数字，指定颜色中的红、绿、蓝和alpha（透明度）的值。这些值是从0（根本没有）到255（最高）的整数。这些RGBA值分配给单个像素，像素是计算机屏幕上能显示一种颜色的最小点（你可以想到，屏幕上有几百万像素）。像素的RGB设置准确地告诉它应该显示哪种颜色的色彩。图像也有一个alpha值，用于生成RGBA值。如果图像显示在屏幕上，遮住了背景图像或桌面墙纸，alpha值决定了“透过”这个图像的象素，你可以看到多少背景。

在Pillow中，RGBA值表示为四个整数值的元组。例如，红色表示为（255，0，0，255）。这种颜色中红的值为最大，没有绿和蓝，并且alpha值最大，这意味着它完全不透明。绿色表示为（0，255，0，255），蓝色是（0，0，255，255）。白色是各种颜色的组合，即

（255，255，255，255），而黑色没有任何颜色，是（0，0，0，255）。

如果颜色的alpha值为0，不论RGB值是什么，该颜色是不可见的。毕竟，不可见的红色看起来就像不可见的黑色一样。

Pillow使用了HTML使用的标准颜色名称。表17-1列出了一些标准颜色的名称和值。

表**17-1** 标准颜色名称及其**RGB**值

名称	**RGBA**值	名称	**RGBA**值
White	(255, 255, 255, 255)	Red	(255, 0, 0, 255)
Green	(0, 128, 0, 255)	Blue	(0, 0, 255, 255)
Gray	(128, 128, 128, 255)	Yellow	(255, 255, 0, 255)
Black	(0, 0, 0, 255)	Purple	(128, 0, 128, 255)

Pillow提供ImageColor.getcolor()函数，所以你不必记住想用的颜色的RGBA值。该函数接受一个颜色名称字符串作为第一个参数，字符串'RGBA'作为第二个参数，返回一个RGBA元组。

要了解该函数的工作方式，就在交互式环境中输入以下代码：

```
❶ >>> from PIL import ImageColor



❷ >>> ImageColor.getcolor('red', 'RGBA')
```

```
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

首先，你需要从PIL导入ImageColor模块❶（不是从Pillow，稍后你就会明白为什么）。传递给ImageColor.getcolor()的颜色名称字符串是不区分大小写的，所以传入'red'❷和传入'RED'❸将得到同样的RGBA元组。还可以传递更多的不常见的颜色名称，如'chocolate'和'Cornflower Blue'。

Pillow支持大量的颜色名称，从'aliceblue'到'whitesmoke'。在http://nostarch. com/automatestuff/ 的资源中，可以找到超过100种标准颜色名称的完整列表。

17.1.2 坐标和Box元组

图像像素用x和y坐标指定，分别指定像素在图像中的水平和垂直位置。原点是位于图像左上角的像素，用符号（0，0）指定。第一个0表示x坐标，它以原点处为0，从左至右增加。第二个0表示y坐标，它以原点处为0，从上至下增加。这值得重复一下：y坐标向下走增加，你可能还记得数学课上使用的y坐标，与此相反。图17-1展示了这个坐标系统的工作方式。

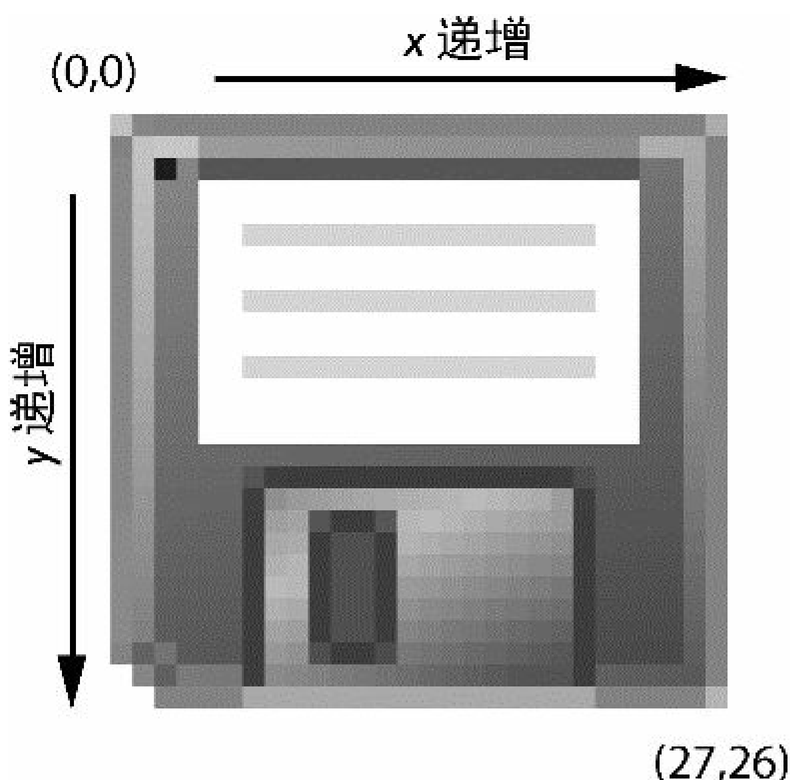


图17-1 27×26的图像的x和y坐标，某种古老的数据存储装置

CMYK和RGB着色

上小学时你学过，混合红、黄、蓝三种颜料可以得到其他颜色。例如，可以混合蓝色和黄色，得到绿色颜料。这就是所谓的减色模型，它适用于染料、油墨和颜料。这就是为什么彩色打印机有的CMYK墨盒：青色（蓝色）、品红色（红色）、黄色和黑色墨水可以混合在一起，形成任何颜色。

然而，光的物理使用所谓的加色模型。如果组合光（例如由计算机屏幕发出的光），红、绿和蓝光可以组合形成其他颜色。这就是为什么在计算机程序中使用RGB值表示颜色。

许多Pillow函数和方法需要一个矩形元组参数。这意味着Pillow需

要一个四个整坐标的元组，表示图像中的一个矩形区域。四个整数按顺序分别是：

- 左：该矩形的最左边的x坐标。
- 顶：该矩形的顶边的y坐标。
- 右：该矩形的最右边右面一个像素的x坐标。此整数必须比左边整数大。
- 底：该矩形的底边下面一个像素的y坐标。此整数必须比顶边整数大。

注意，该矩形包括左和顶坐标，直到但不包括右和底坐标。例如，矩形元组（3, 1, 9, 6）表示图17-2中黑色矩形的所有像素。

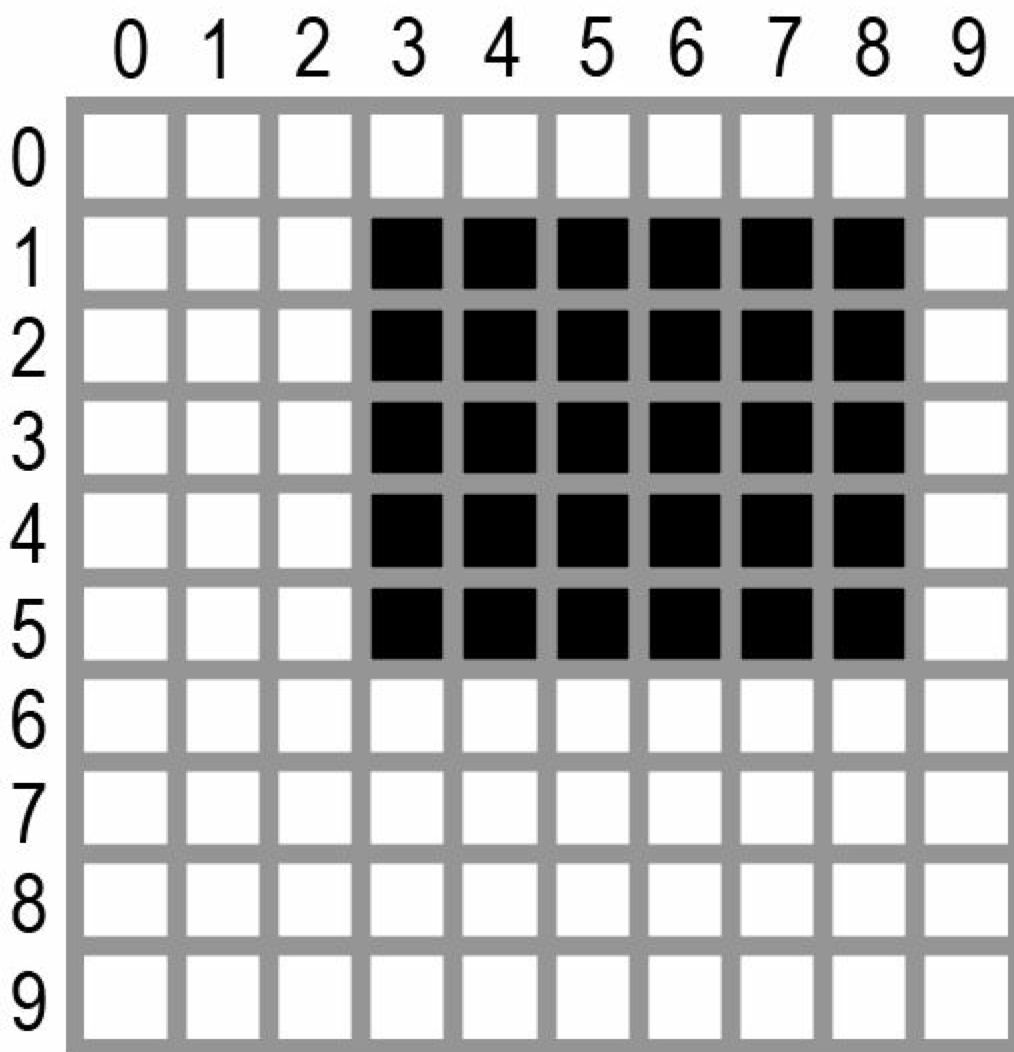


图17-2 由矩形元组（3, 1, 9, 6）表示的区域

17.2 用**Pillow**操作图像

既然知道了 Pillow 中颜色和坐标的工作方式，就让我们用 Pillow 来处理图像。图 17-3 中的图像将用于本章中所有交互式环境的例子。你可以从http://nostarch. com/automatestuff/ 下载它。



图17-3 我的猫Zophie。照片上看起来增加了10磅（对猫来说很多）

将图像文件Zophie.png放在当前工作目录中，你就可以将Zophie的图像加载到Python中，像这样：

```
>>> from PIL import Image

>>> catIm = Image.open('zophie.png')
```

要加载图像，就从Pillow导入Image模块，并调用Image.open()，传入图像的文件名。然后，可以将加载图像保存在CatIm这样的变量中。Pillow的模块名称是PIL，这保持与老模块Python Imaging Library向后兼容，这就是为什么必须from PIL import Image，而不是from Pillow import Image。由于Pillow的创建者设计Pillow模块的方式，你必须使用from PIL import Image形式的import语句，而不是简单地import PIL。

如果图像文件不在当前工作目录，就调用os.chdir()函数，将工作目录变为包含图像文件的文件夹。

```
>>> import os

>>> os.chdir('C:\\folder_with_image_file')
```

Image.open()函数的返回值是Image对象数据类型，它是Pillow将图像表示为Python值的方法。可以调用Image.open()，传入文件名字符串，从一个图像文件（任何格式）加载一个Image对象。通过save()方法，对Image对象的所有更改都可以保存到图像文件中（也是任何格式）。所有的旋转、调整大小、裁剪、绘画和其他图像操作，都通过这个Image对象上的方法调用来完成。

为了让本章的例子更简短，我假定你已导入了Pillow的Image模块，并将Zophie的图像保存在变量catIm中。要确保zophie.png文件在当前工作目录中，让Image.open()函数能找到它。否则，必须在Image.open()的字符串参数中指定完整的绝对路径。

17.2.1 处理Image数据类型

Image对象有一些有用的属性，提供了加载的图像文件的基本信息：它的宽度和高度、文件名和图像格式（如JPEG、GIF或PNG）。

例如，在交互式环境中输入以下代码：

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
>>> catIm.size
❶ (816, 1088)
```

```
❷ >>> width, height = catIm.size
❸ >>> width
  816
❹ >>> height
  1088
  >>> catIm.filename
  'zophie.png'
  >>> catIm.format
  'PNG'
  >>> catIm.format_description
  'Portable network graphics'
❺ >>> catIm.save('zophie.jpg')
```

从Zophie.png得到一个Image对象并保存在catIm中后，我们可以看到该对象的size属性是一个元组，包含该图像的宽度和高度的像素数❶。我们可以将元组中的值赋给width和height变量❷，以便分别访问宽度❸和高度❹。filename属性描述了原始文件的名称。format和format_description属性是字符串，描述了原始文件的图像格式（format_description比较详细）。

最后，调用save()方法，传入'zophie.jpg'，将新图像以文件名zophie.jpg保存到硬盘上❺。Pillow看到文件扩展名是jpg，就自动使用JPEG图像格式来保存图像。现在硬盘上应该有两个图像，zophie.png和zophie.jpg。虽然这些文件都基于相同的图像，但它们不一样，因为格式不同。

Pillow 还提供了 Image.new()函数，它返回一个 Image 对象。这很像Image.open()，不过Image.new()返回的对象表示空白的图像。Image.new()的参数如下：

- 字符串'RGBA'，将颜色模式设置为RGBA（还有其他模式，但本书没有涉及）。
- 大小，是两个整数元组，作为新图像的宽度和高度。
- 图像开始采用的背景颜色，是一个表示RGBA值的四整数元组。你可以用ImageColor.getcolor()函数的返回值作为这个参数。另外，Image.new()也支持传入标准颜色名称的字符串。

例如，在交互式环境中输入以下代码：

```
  >>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
```

```
>>> im.save('purpleImage.png')

❷ >>> im2 = Image.new('RGBA', (20, 20))

>>> im2.save('transparentImage.png')
```

这里，我们创建了一个Image对象，它有100像素宽、200像素高，带有紫色背景❶。然后，该图像存入文件purpleImage.png中。我们再次调用Image.new()，创建另一个Image对象，这次传入（20, 20）作为大小，没有指定背景色❷。如果未指定颜色参数，默认的颜色是不可见的黑色（0，0，0，0），因此第二个图像具有透明背景，我们将这个20×20的透明正方形存入transparentImage.png。

17.2.2 裁剪图片

裁剪图像是指在图像内选择一个矩形区域，并删除矩形之外的一切。Image对象的crop()方法接受一个矩形元组，返回一个Image对象，表示裁剪后的图像。裁剪不是在原图上发生的，也就是说，原始的Image对象原封不动，crop()方法返回一个新的Image对象。请记住，矩形元组（这里就是要裁剪的区域）包括左列和顶行的像素，直至但不包括右列和底行的像素。

在交互式环境中输入以下代码：

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))
>>> croppedIm.save('cropped.png')
```

这得到一个新的Image对象，是剪裁后的图像，保存在croppedIm中，然后调用croppedIm的save()，将裁剪后的图像存入cropped.png。新文件cropped.png从原始图像创建，如图17-4所示。

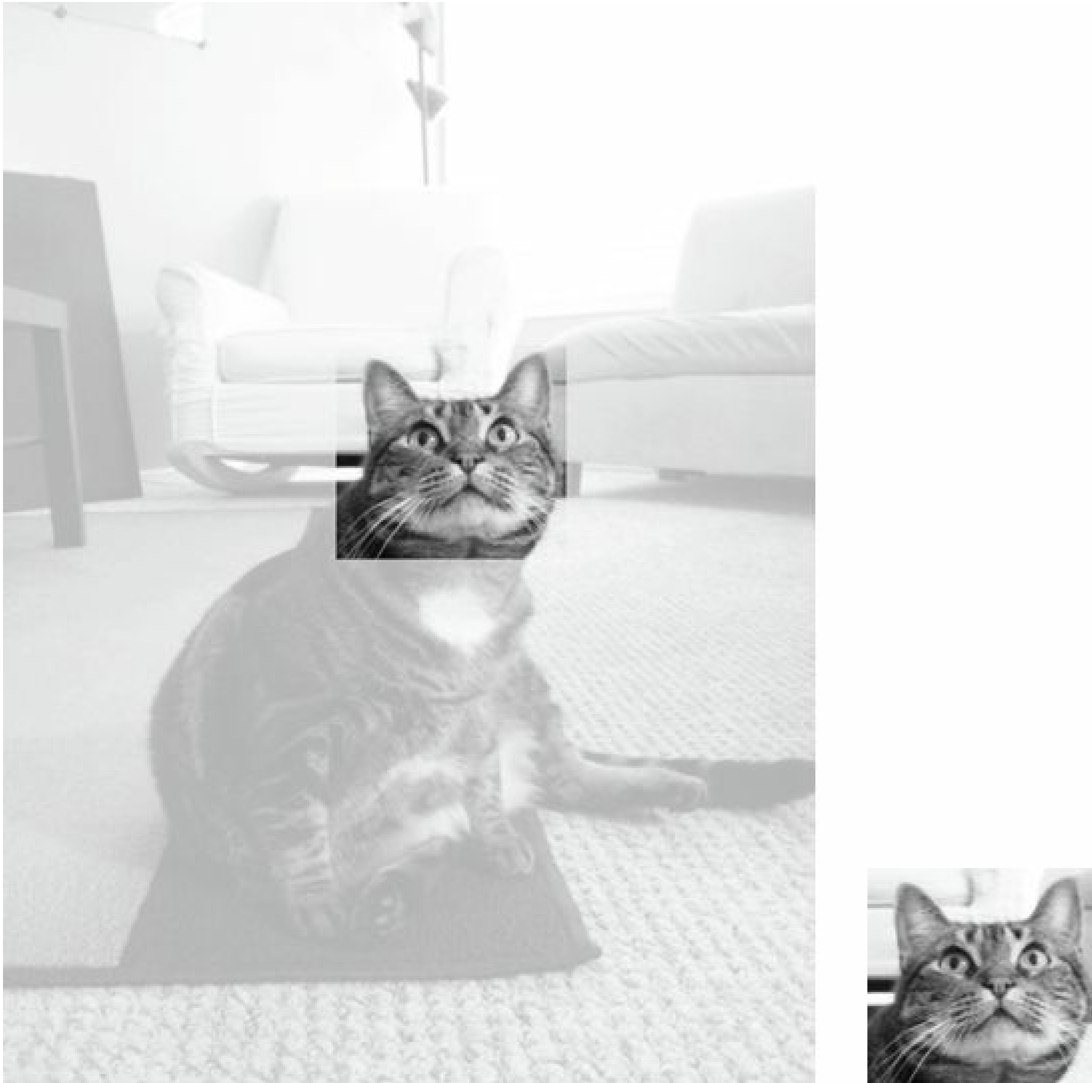


图17-4 新图像只有原始图像剪裁后的部分

17.2.3 复制和粘贴图像到其他图像

copy()方法返回一个新的Image对象，它和原来的Image对象具有一样的图像。如果需要修改图像，同时也希望保持原有的版本不变，这非常有用。例如，在交互式环境中输入以下代码：

```
>>> catIm = Image.open('zophie.png')
```

```
>>> catCopyIm = catIm.copy()
```

catIm和catCopyIm变量包含了两个独立的Image对象，它们的图像相同。既然catCopyIm中保存了一个Image对象，你可以随意修改catCopyIm，将它存入一个新的文件名，而zophie.png没有改变。例如，让我们尝试用paste()方法修改catCopyIm。

paste()方法在Image对象调用，将另一个图像粘贴在它上面。我们继续交互式环境的例子，将一个较小的图像粘贴到catCopyIm。

```
>>> faceIm = catIm.crop((335, 345, 565, 560))
>>> faceIm.size
(230, 215)
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

首先我们向crop()传入一个矩形元组，指定zophie.png中的一个矩形区域，包含Zophie的脸。这将创建一个Image对象，表示230×215的剪裁区域，保存在faceIm中。现在，我们可以将faceIm粘贴到catCopyIm。paste()方法有两个参数：一个“源”Image对象，一个包含x和y坐标的元组，指明源Image对象粘贴到主Image对象时左上角的位置。这里，我们在catCopyIm上两次调用paste()，第一次传入(0, 0)，第二次传入(400, 500)。这将faceIm两次粘贴到catCopyIm：一次faceIm的左上角在(0, 0)，一次faceIm的左上角在(400, 500)。最后，我们将修改后的catCopyIm存入pasted.png。pasted.png如图17-5所示。

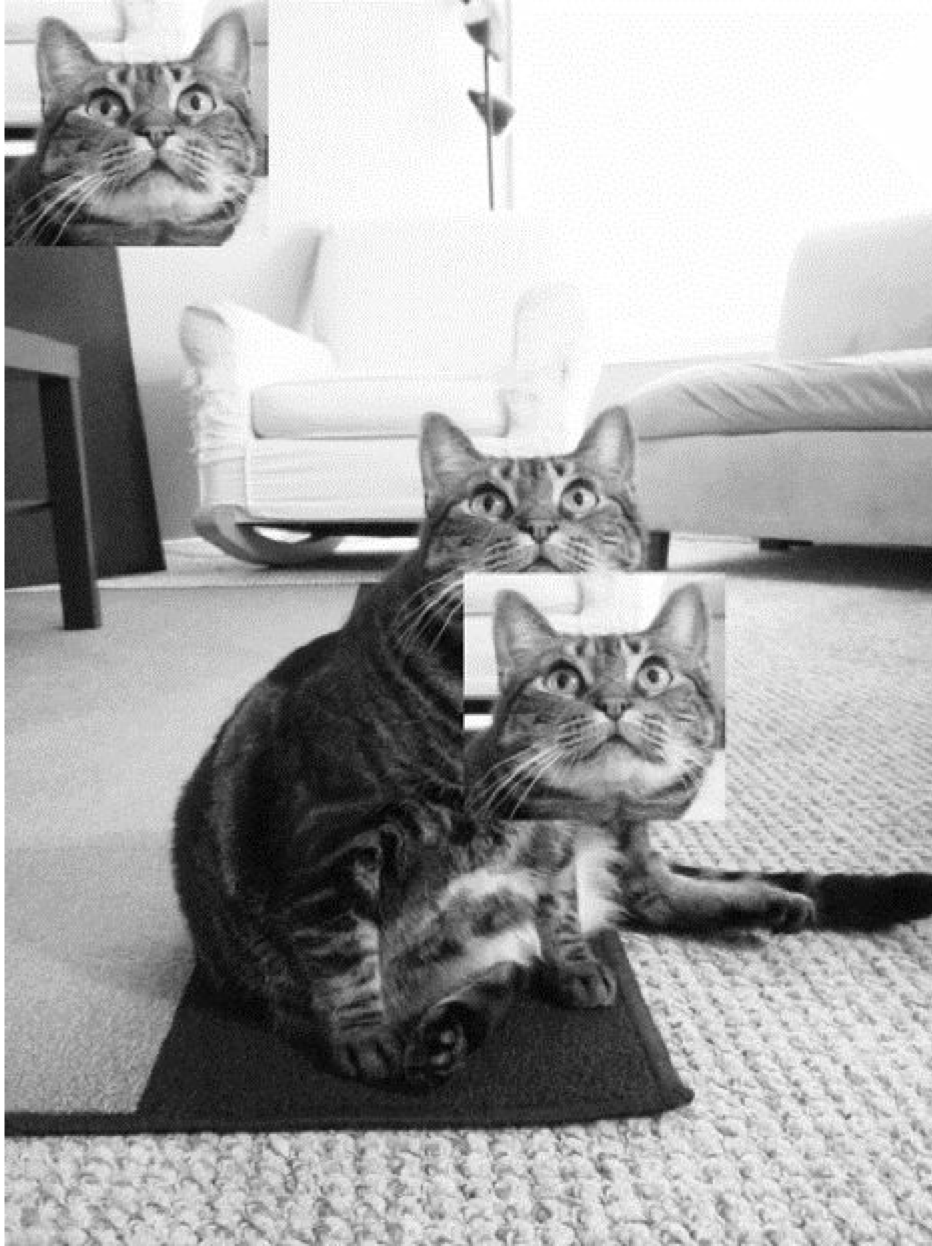


图17-5 Zophie猫，包含两次粘贴她的脸

注意

尽管名称是copy()和paste()，但Pillow中的方法不使用计算机的剪贴板。

请注意，paste()方法在原图上修改它的Image对象，它不会返回粘贴后图像的Image对象。如果想调用paste()，但还要保持原始图像的未修改版本，就需要先复制图像，然后在副本上调用paste()。

假定要用Zophie的头平铺整个图像，如图17-6所示。可以用两个for

循环来实现这个效果。继续交互式环境的例子，输入以下代码：

```
  >>> catImWidth, catImHeight = catIm.size

  >>> faceImWidth, faceImHeight = faceIm.size

❶ >>> catCopyTwo = catIm.copy()

❷ >>> for left in range(0, catImWidth, faceImWidth):

❸           for top in range(0, catImHeight, faceImHeight):

              print(left, top)

              catCopyTwo.paste(faceIm, (left, top))

  0 0
  0 215
  0 430
```

```
0 645
0 860
0 1075
230 0
230 215
--snip
--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')
```

这里，我们将catIm的高度的宽度保存在catImWidth和catImHeight中。在❶，我们得到了catIm的副本，并保存在catCopyTwo。既然有了一个副本可以粘贴，我们就开始循环，将faceIm粘贴到catCopyTwo。外层for循环的left变量从0开始，增量是faceImWidth（即230）❷。内层for循环的top变量从0开始，增量是faceImHeight（即215）❸。这些嵌套的for循环生成了left和top的值，将faceIm图像按照网格粘贴到Image对象catCopyTwo，如图17-6所示。为了看到嵌套循环的工作，我们打印了left和top。粘贴完成后，我们将修改后的catCopyTwo保存到tiled.png。



图17-6 嵌套的for循环与paste()，用于复制猫脸（可以称之为dupli-cat）

17.2.4 调整图像大小

resize()方法在Image对象上调用，返回指定宽度和高度的一个新Image对象。它接受两个整数的元组作为参数，表示返回图像的新高度和宽度。在交互式环境中输入以下代码：

```
❶ >>> width, height = catIm.size
```

```
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))

   >>> quartersizedIm.save('quartersized.png')

❸ >>> svelteIm = catIm.resize((width, height + 300))

   >>> svelteIm.save('svelte.png')
```

这里，我们将catIm.size元组中的两个值赋给变量width和height❶。使用width和height，而不是catIm.size[0]和catIm.size[1]，让接下来的代码更易读。

第一个resize()调用传入int(width / 2)作为新宽度，int(height / 2)作为新高度❷，所以resize()返回的Image对象具有原始图像的一半长度和宽度，是原始图像大小的四分之一。resize()方法的元组参数中只允许整数，这就是为什么需要用int()调用对两个除以2的值取整。

这个大小调整保持了相同比例的宽度和高度。但传入resize()的新宽度和高度不必与原始图像成比例。svelteIm变量保存了一个Image对象，宽度与原始图像相同，但高度增加了300像素❸，让Zophie显得更苗条。

请注意，resize()方法不会在原图上修改Image对象，而是返回一个新的Image对象。

粘贴透明像素

通常透明像素像白色像素一样粘贴。如果要粘贴图像有透明像素，就将该图像作为第三个参数传入，这样就不会粘贴一个不透明的矩形。这个第三参数是“遮罩”Image对象。遮罩是一个Image对象，其中alpha值是有效的，但红、绿、蓝值将被忽略。遮罩告诉paste()函数哪些像素应该复制，哪些应该保持透明。遮罩的高级用法超出了本书的范围，但如果你想粘贴有透明像素的图像，就再传入该Image对象作为第三个参数。

17.2.5 旋转和翻转图像

图像可以用rotate()方法旋转，该方法返回旋转后的新Image对象，并保持原始Image对象不变。rotate()的参数是一个整数或浮点数，表示图像逆时针旋转的度数。在交互式环境中输入以下代码：

```
>>> catIm.rotate(90).save('rotated90.png')
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

注意，可以连续调用方法，对rotate()返回的Image对象直接调用save()。第一个rotate()和save()调用得到一个逆时针旋转90度的新Image对象，并将旋转后的图像存入rotated90.png。第二和第三个调用做的事情一样，但旋转了180度和270度。结果如图17-7所示。

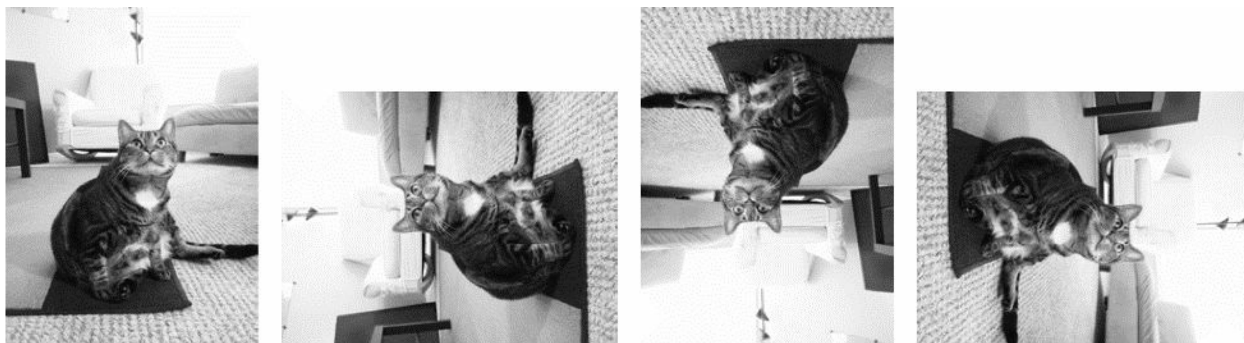


图17-7 原始图像（左）和逆时针旋转90度、180度和270度的图像

注意，当图像旋转90度或270度时，宽度和高度会变化。如果旋转其他角度，图像的原始尺寸会保持。在Windows上，使用黑色的背景来填补旋转造成的缝隙，如图17-8所示。在OS X上，使用透明的像素来填补缝隙。rotate()方法有一个可选的expand关键字参数，如果设置为True，就会放大图像的尺寸，以适应整个旋转后的新图像。例如，在交互式环境中输入以下代码：

```
>>> catIm.rotate(6).save('rotated6.png')
>>> catIm.rotate(6, expand=True).save('rotated6_expanded.png')
```

第一次调用将图像旋转6度，并存入rotate.png（参见图17-8的左边的图像）。第二次调用将图像旋转6度，expand设置为True，并存入rotate6_expanded.png（参见图17-8的右侧的图像）。

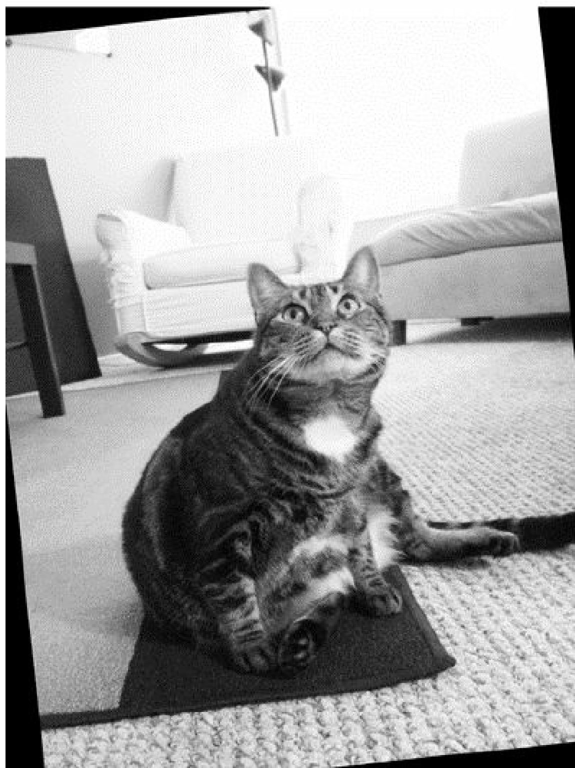
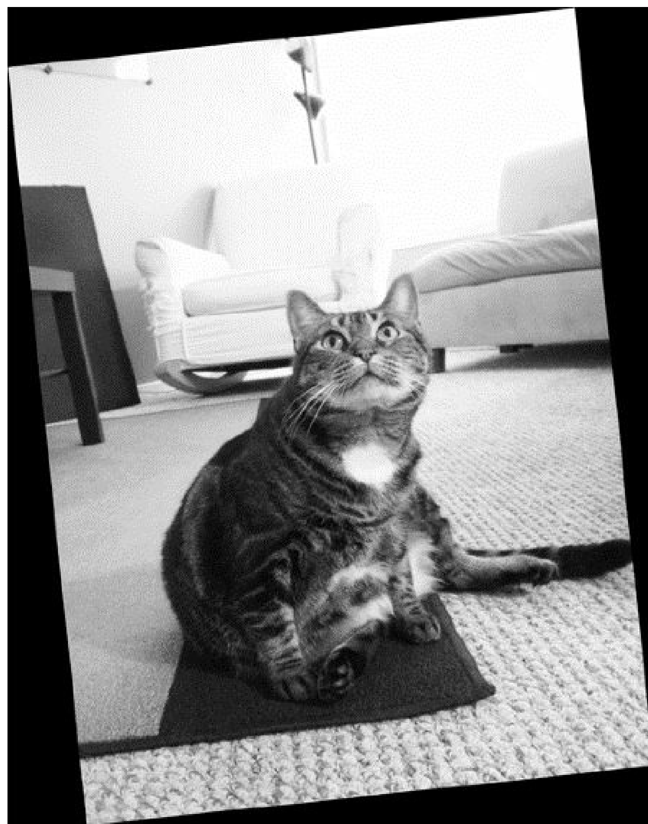



图17-8 图像普通旋转6度（左），以及使用expand=True（右）

利用transpose()方法，还可以得到图像的“镜像翻转”。必须向transpose()方法传入Image.FLIP_LEFT_RIGHT或Image.FLIP_TOP_BOTTOM。在交互式环境中输入以下代码：

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

像rotate()一样，transpose()会创建一个新Image对象。这里我们传入Image.FLIP_ LEFT_RIGHT，让图像水平翻转，然后存入horizontal_flip.png。要垂直翻转图像，传入Image.FLIP_TOP_BOTTOM，并存入vertical_flip.png。结果如图17-9所示。

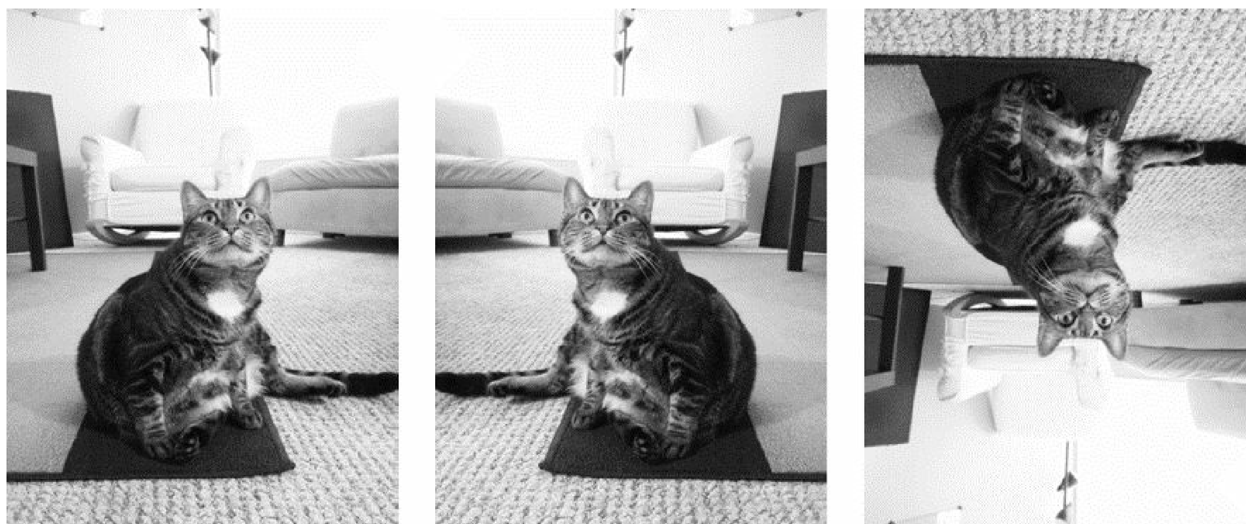


图17-9 原始图像（左），水平翻转（中），垂直翻转（右）

17.2.6 更改单个像素

单个像素的颜色可以通过getpixel()和putpixel()方法取得和设置。它们都接受一个元组，表示像素的x和y坐标。putpixel()方法还接受一个元组，作为该像素的颜色。这个颜色参数是四整数RGBA元组或三整数RGB元组。在交互式环境中输入以下代码：

```
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
  (0, 0, 0, 0)
```

```
❸ >>> for x in range(100):
          for y in range(50):
❹             im.putpixel((x, y), (210, 210, 210))

  >>> from PIL import ImageColor
❺ >>> for x in range(100):
          for y in range(50, 100):
❻             im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
```

```
>>> im.getpixel((0, 0))
(210, 210, 210, 255)
>>> im.getpixel((0, 50))
(169, 169, 169, 255)
>>> im.save('putPixel.png')
```

在❶，我们得到一个新图像，这是一个100×100的透明正方形。对一些坐标调用getPixel()将返回（0，0，0，0），因为图像是透明的❷。要给图像中的像素上色，我们可以使用嵌套的for循环，遍历图像上半部分的所有像素❸，用putpixel()设置每个像素的颜色❹。这里我们向putpixel()传入RGB元组（210，210，210），即浅灰色。

假定我们希望图像下半部分是暗灰色，但不知道深灰色的RGB元组。putpixel()方法不接受'darkgray'这样的标准颜色名称，所以必须使用ImageColor.getcolor()来获得'darkgray'的颜色元组。循环遍历图像的下半部分像素❺，向putpixel()传入ImageColor. getcolor()的返回值❻，你现在应该得到一个图像，上半部分是浅灰色，下半部分是深灰色，如图17-10所示。可以对一些坐标调用getPixel()，确认指定像素的颜色符合你的期望。最后，将图像存入putPixel.png。

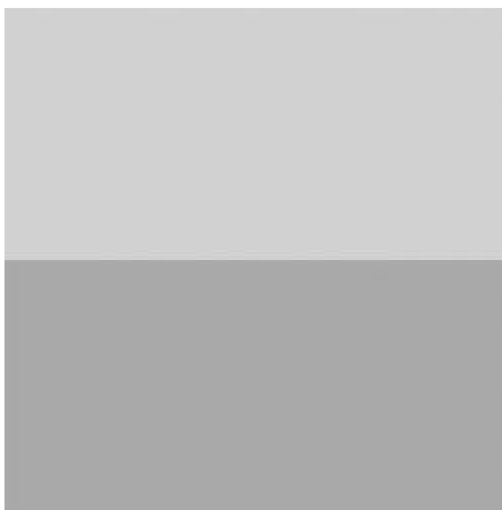


图17-10 putPixel.png中的图像

当然，在图像上一次绘制一个像素不是很方便。如果需要绘制形状，就使用本章稍后介绍的ImageDraw函数。

17.3 项目：添加徽标

假设你有一项无聊的工作，要调整数千张图片的大小，并在每张图片的角上增加一个小徽标水印。使用基本的图形程序，如Paintbrush或Paint，完成这项工作需要很长时间。像Photoshop这样神奇的应用程序可以批量处理，但这个软件要花几百美元。让我们写一个脚本来完成工作。

假定图 17-11 是要添加到每个图像右下角的标识：带有白色边框的黑猫图标，图像的其余部分是透明的。

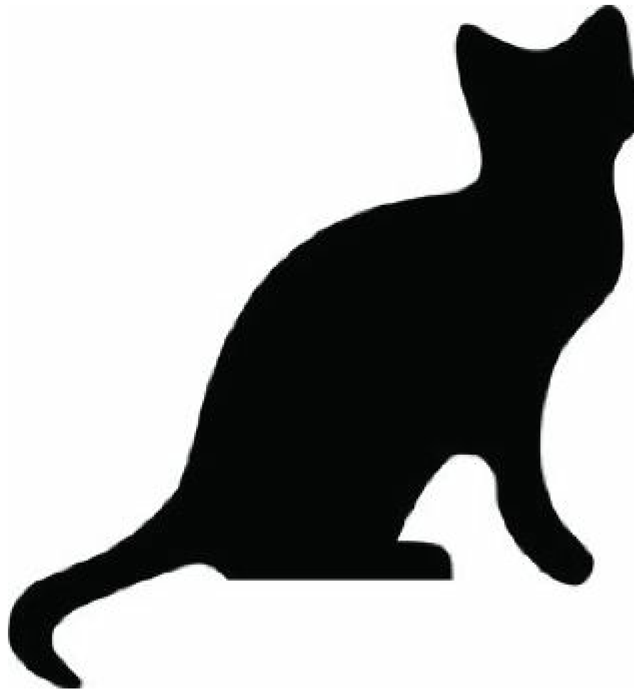


图17-11 添加到图像中的徽标

总的来说，程序应该完成下面的事：

- 载入徽标图像。
- 循环遍历工作目标中的所有.png和.jpg文件。
- 检查图片是否宽于或高于300像素。
- 如果是，将宽度或高度中较大的一个减小为300像素，并按比例缩小的另一维度。
- 在角上粘贴徽标图像。
- 将改变的图像存入另一个文件夹。

这意味着代码需要做到以下几点：

- 打开catlogo.png文件作为Image对象。
- 循环遍历os.listdir('.')返回的字符串。
- 通过size属性取得图像的宽度和高度。
- 计算调整后图像的新高度和宽度。
- 调用resize()方法来调整图像大小。
- 调用paste()方法来粘贴徽标。
- 调用save()方法保存更改，使用原来的文件名。

第1步：打开徽标图像

针对这个项目，打开一个新的文件编辑器窗口，输入以下代码，并保存为resizeAndAddLogo.py：

```
  #! python3
  # resizeAndAddLogo.py - Resizes all images in current working directory t
  # in a 300x300 square, and adds catlogo.png to the lower-right corner.
  import os
  from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

  # TODO: Loop over all files in the working directory.

  # TODO: Check if image needs to be resized.

  # TODO: Calculate the new width and height to resize to.

  # TODO: Resize the image.

  # TODO: Add the logo.

  # TODO: Save changes.
```

在程序开始时设置SQUARE_FIT_SIZE❶和LOGO_FILENAME❷常量，这让程序以后更容易修改。假定你要添加的徽标不是猫图标，或者假定将输出图像的最大大小要减少的值不是300像素。有了程序开始时定义的这些常量，你可以打开代码，修改一下这些值，就大功告成了（或者你可以让这些常量的值从命令行参数获得）。没有这些常数，就要在代码中寻找所有的300和'catlogo.png'，将它们替换新项目的值。总之，使用常量使程序更加通用。

徽标Image对象从Image.open()返回❸。为了增强可读性，logoWidth和logoHeight被赋予logoIm.size中的值❹。

该程序的其余部分目前是TODO注释，说明了程序的骨架。

第2步：遍历所有文件并打开图像

现在，需要找到当前工作目录中的每个PNG文件和.jpg文件。请注意，你不希望将徽标图像添加到徽标图像本身，所以程序应该跳过所有像LOGO_FILENAME这样的图像文件名。在程序中添加以下代码：

```
  #! python3
  # resizeAndAddLogo.py - Resizes all images in current working directory t
  # in a 300x300 square, and adds catlogo.png to the lower-right corner.

  import os
  from PIL import Image

  --snip

--
  os.makedirs('withLogo', exist_ok=True)
  # Loop over all files in the working directory.

❶ for filename in os.listdir('.'):

❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \

        or filename == LOGO_FILENAME:
```

```
❸        continue     # skip non-image files and the logo file itself

❹     im = Image.open(filename)

      width, height = im.size

  --snip

--
```

首先，os.makedirs()调用创建了一个文件夹withLogo，用于保存完成的、带有徽标的图像，而不是覆盖原始图像文件。关键字参数exist_ok=True将防止os.makedirs()在withLogo已存在时抛出异常。在用os.listdir('.')遍历工作目录中的所有文件时❶，较长的if语句❷检查每个filename是否以.png或.jpg结束。如果不是，或者该文件是徽标图像本身，循环就跳过它，使用continue❸去处理下一个文件。如果filename确实以'.png'或'.jpg'结束（而且不是徽标文件），可以将它打开为一个Image对象❹，并设置width和height。

第3步：调整图像的大小

只在有宽或高超过SQUARE_FIT_SIZE时（在这个例子中，是300像

素），该程序才应该调整图像的大小，所以将所有大小调整的代码放在一个检查width和height变量的if语句内。在程序中添加以下代码：

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory t
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip

--

    # Check if image needs to be resized.

    if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:

        # Calculate the new width and height to resize to.

        if width > height:

❶             height = int((SQUARE_FIT_SIZE / width) * height)
```

```
            width = SQUARE_FIT_SIZE
        else:
❷           width = int((SQUARE_FIT_SIZE / height) * width)
            height = SQUARE_FIT_SIZE

        # Resize the image.
        print('Resizing %s...' % (filename))
❸       im = im.resize((width, height))

--snip
```

```
--
```

如果图像确实需要调整大小，就需要弄清楚它是太宽还是太高。如果width大于height，则高度应该根据宽度同比例减小❶。这个比例是当前宽度除以SQUARE_ FIT_SIZE的值。新的高度值是这个比例乘以当前高度值。由于除法运算符返回一个浮点值，而resize()要求的尺寸是整数，所以要记得将结果用int()函数转换成整数。最后，新的width值就设置为SQUARE_FIT_SIZE。

如果height大于或等于width（这两种情况都在else子句中处理），那么进行同样的计算，只是交换height和width变量的位置❷。

在width和height包含新图像尺寸后，将它们传入resize()方法，并返回的Image对象保存在im中❸。

第4步：添加徽标，并保存更改

不论图像是否调整大小，徽标仍应粘贴到右下角。徽标粘贴的确切位置取决于图像的大小和徽标的大小。图 17-12 展示了如何计算粘贴的位置。粘贴徽标的左坐标将是图像宽度减去徽标宽度，顶坐标将是图像高度减去徽标高度。

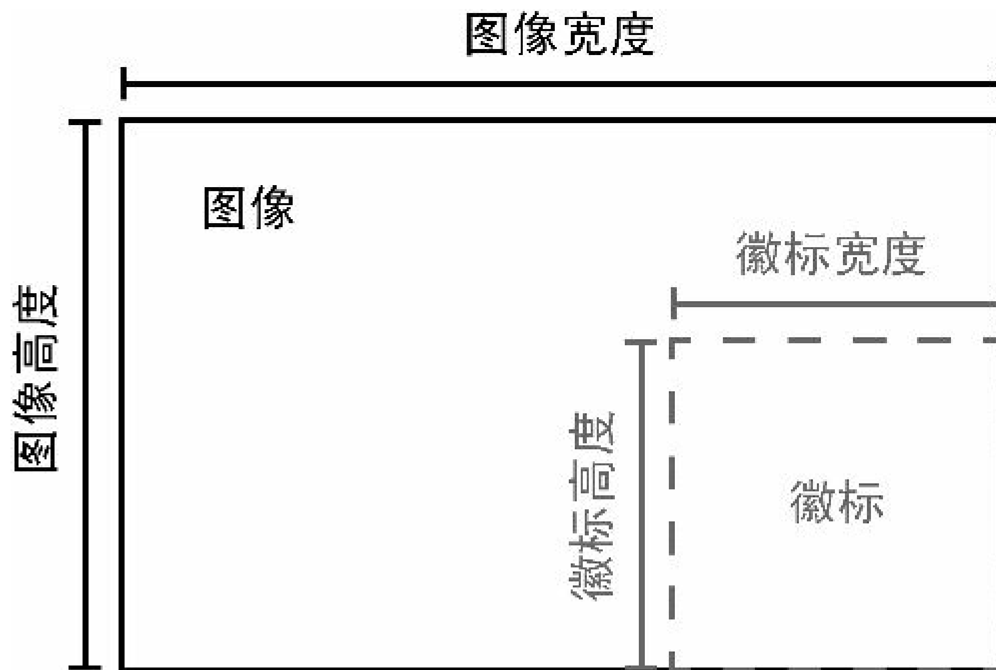


图17-12 在右下角放置徽标的左坐标和顶坐标，应该是图像的宽度/高度减去徽标宽度/高度

代码将徽标粘贴到图像中后，应保存修改后的Image对象。将以下代码添加到程序中：

```
  #! python3
  # resizeAndAddLogo.py - Resizes all images in current working directory t
  # in a 300x300 square, and adds catlogo.png to the lower-right corner.

  import os
  from PIL import Image
  --snip






--








# Check if image needs to be resized.
```

```
--snip

--
      # Add the logo.

❶     print('Adding logo to %s...' % (filename))

❷     im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)

      # Save changes.

❸     im.save(os.path.join('withLogo', filename))
```

新的代码输出一条消息，告诉用户徽标已被加入❶，将logoIm粘贴到im中计算的坐标处❷，并将变更保存到withLogo目录的filename中

❸。如果运行这个程序，zophie.png文件是工作目录中唯一的图像，输出会是这样：

```
Resizing zophie.png...
Adding logo to zophie.png...
```

图像zophie.png将变成225×300像素的图像，如图17-13所示。请记住，如果没有传入logoIm作为第三个参数，paste()方法不会粘贴透明的像素。这个程序可以在短短几分钟内自动调整几百幅图像，并“加上徽标”。

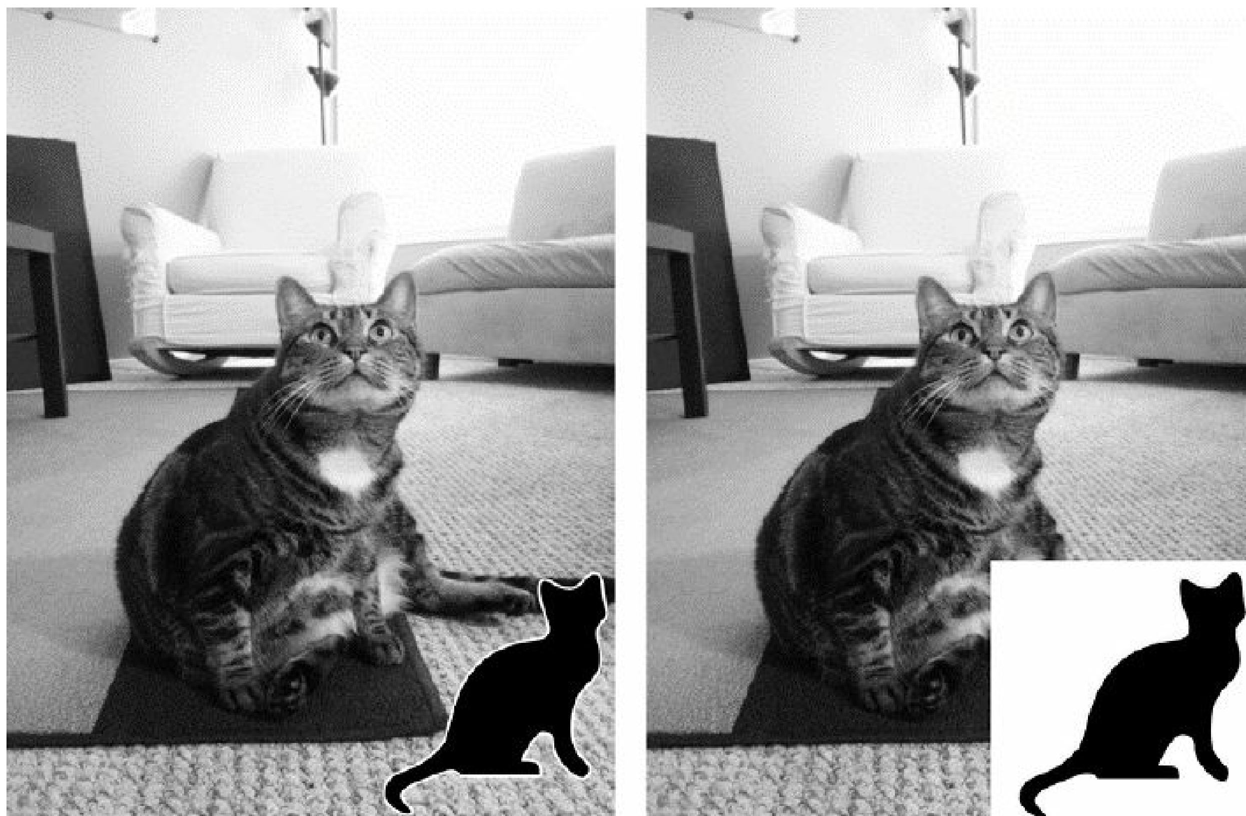


图17-13 图像zophie.png调整了大小并加上了徽标（左）。如果忘记了第三个参数，徽标中透明的像素将被复制为不透明的白色像素（右）

第5步：类似程序的想法

能够批量合成图像或修改图像大小，在许多应用中都有用。可以编写类似的程序，完成以下任务：

- 为图像添加文字或网站URL。
- 为图像添加时间戳。
- 根据图像的大小，将图像复制或移动到不同的文件夹中。
- 为图像添加一个几乎透明的水印，防止他人复制。

17.4 在图像上绘画

如果需要在图像上画线、矩形、圆形或其他简单形状，就用Pillow的ImageDraw模块。在交互式环境中输入以下代码：

```
>>> from PIL import Image, ImageDraw




>>> im = Image.new('RGBA', (200, 200), 'white')




>>> draw = ImageDraw.Draw(im)
```

首先，我们导入Image和ImageDraw。然后，创建一个新的图像，在这个例子中，是200×200的白色图像，将这个Image对象保存在Im中。我们将该Image对象传入ImageDraw.Draw()函数，得到一个ImageDraw对象。这个对象有一些方法，可以在Image对象上绘制形状和文字。将ImageDraw对象保存在变量draw中，这样就能在接下来的例子中方便地

使用它。

17.4.1 绘制形状

下面的ImageDraw方法在图像上绘制各种形状。这些方法的fill和outline参数是可选的，如果未指定，默认为白色。

点

point(xy, fill)方法绘制单个像素。xy参数表示要画的点的列表。该列表可以是x和y坐标的元组的列表，例如[(x, y), (x, y), ...]，或是没有元组的x和y坐标的列表，例如[x1, y1, x2, y2, ...]。fill参数是点的颜色，要么是一个RGBA元组，要么是颜色名称的字符串，如'red'。fill参数是可选的。

线

line(xy, fill, width)方法绘制一条线或一系列的线。xy要么是一个元组的列表，例如[(x, y), (x, y), ...]，要么是一个整数列表，例如[x1, y1, x2, y2, ...]。每个点都是正在绘制的线上的一个连接点。可选的fill参数是线的颜色，是一个RGBA元组或颜色名称。可选的width参数是线的宽度，如果未指定，缺省值为1。

矩形

rectangle(xy, fill, outline)方法绘制一个矩形。xy参数是一个矩形元组，形式为(left, top, right, bottom)。left和top值指定了矩形左上角的x和y坐标，right和bottom指定了矩形的右下角。可选的fill参数是颜色，将填充该矩形的内部。可选的outline参数是矩形轮廓的颜色。

椭圆

ellipse(xy, fill, outline)方法绘制一个椭圆。如果椭圆的宽度和高度一样，该方法将绘制一个圆。xy参数是一个矩形元组(left, top, right, bottom)，它表示正好包含该椭圆的矩形。可选的fill参数是椭圆内的颜色，可选的outline参数是椭圆轮廓的颜色。

多边形

polygon(xy, fill, outline)方法绘制任意的多边形。xy参数是一个元组列表，例如[(x, y), (x, y), ...]，或者是一个整数列表，例如[x1, y1, x2, y2, ...]，表示多边形边的连接点。最后一对坐标将自动连接到第一对坐标。可选的fill参数是多边形内部的颜色，可选的outline参数是多边形轮廓的颜色。

绘制示例

在交互式环境中输入以下代码：

```
>>> from PIL import Image, ImageDraw

>>> im = Image.new('RGBA', (200, 200), 'white')

>>> draw = ImageDraw.Draw(im)

❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='bla

❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')

❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')
```

```
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)),
  fill='brown')
❺ >>> for i in range(100, 200, 10):
          draw.line([(i, 0), (200, i - 100)], fill='green')

  >>> im.save('drawing.png')
```

为200×200的白色图像生成Image对象后，将它传入ImageDraw.Draw()，获得ImageDraw对象，将它保存在draw中，可以对draw调用绘图方法。这里，我们在图像边缘画上窄的黑色轮廓❶；一个蓝色的矩形，左上角在（20, 30），右下角在（60, 60）❷；一个红色的

椭圆，由（120, 30）到（160, 60）的矩形来定义❸；一个棕色的多边形，有五个顶点❹，以及一些绿线的图案，用for循环绘制❺。得到的drawing.png文件如图17-14所示。

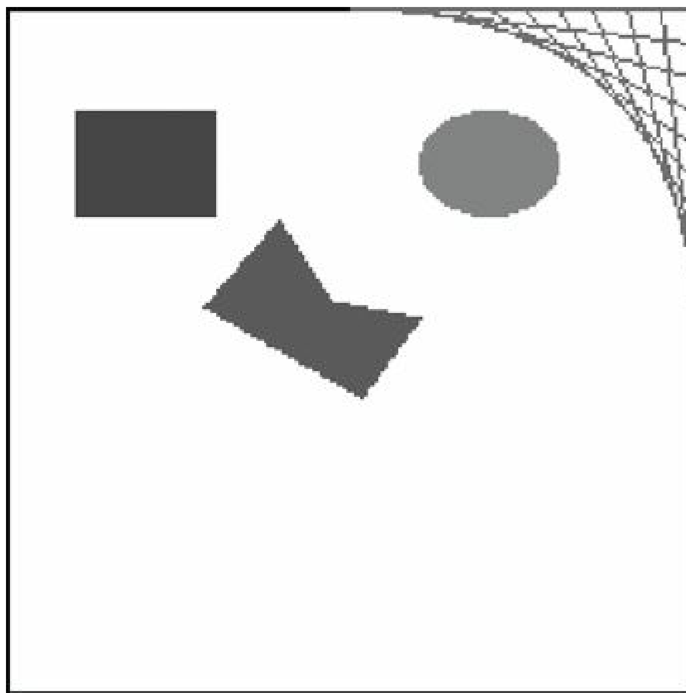


图17-14 得到的图像drawing.png

ImageDraw对象还有另外几个绘制形状的方法。完整的文档在http://pillow. readthedocs.org/en/latest/reference/ImageDraw.html 。

17.4.2 绘制文本

ImageDraw对象还有text()方法，用于在图像上绘制文本。text()方法有4个参数：xy、text、fill和font。

- xy参数是两个整数的元组，指定文本区域的左上角。
- text参数是想写入的文本字符串。
- 可选参数fill是文本的颜色。
- 可选参数font是一个ImageFont对象，用于设置文本的字体和大小。下一节中更详细地介绍了这个参数。

因为通常很难预先知道一块文本在给定的字体下的大小，所以ImageDraw模块也提供了textsize()方法。它的第一个参数是要测量的文

本字符串，第二个参数是可选的ImageFont对象。textsize()方法返回一个两整数元组，表示如果以指定的字体写入图像，文本的宽度和高度。可以利用这个宽度和高度，帮助你精确计算文本放在图像上的位置。

text()的前三个参数非常简单。在用text()向图像绘制文本之前，让我们来看看可选的第四个参数，即ImageFont对象。

text()和textsize()都接受可选的ImageFont对象，作为最后一个参数。要创建这种对象，先执行以下命令：

```
>>> from PIL import ImageFont
```

既然已经导入Pillow的ImageFont模块，就可以调用ImageFont.truetype()函数，它有两个参数。第一个参数是字符串，表示字体的TrueType文件，这是硬盘上实际的字体文件。TrueType字体文件具有.TTF文件扩展名，通常可以在以下文件夹中找到：

- 在Windows上：C:\Windows\Fonts。
- 在OS X上：/Library/Fonts and /System/Library/Fonts。
- 在Linux上：/usr/share/fonts/truetype。

实际上并不需要输入这些路径作为TrueType字体文件的字符串的一部分，因为Python知道自动在这些目录中搜索字体。如果无法找到指定的字体，Python会显示错误。

ImageFont.truetype()的第二个参数是一个整数，表示字体大小的点数（而不是像素）。请记住，Pillow创建的PNG图像默认是每英寸72像素，一点是1/72英寸。

在交互式环境中输入以下代码，用你的操作系统中实际的文件夹名称替换FONT_FOLDER：

```
  >>> from PIL import Image, ImageDraw, ImageFont

  >>> import os

❶ >>> im = Image.new('RGBA', (200, 200), 'white')

❷ >>> draw = ImageDraw.Draw(im)

❸ >>> draw.text((20, 150), 'Hello', fill='purple')

  >>> fontsFolder = 'FONT_FOLDER' # e.g. ‘/Library/Fonts’

❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf')

❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
```

```
>>> im.save('text.png')
```

导入Image、ImageDraw、ImageFont和os后，我们生成一个Image对象，是新的200×200白色图像❶，并通过这个Image对象得到一个ImageDraw对象❷。我们使用text()在（20, 150）以紫色绘制Hello❸。在这次text()调用中，我们没有传入可选的第四个参数，所以这段文本的字体和大小没有定制。

要设置字体和大小，我们首先将文件夹名称（如/Library/Fonts）保存在fontsFolder中。然后调用ImageFont.truetype()，传入我们想要的字体的.TTF文件，之后是表示字体大小的整数❹。将ImageFont.truetype()返回的Font对象保存在arialFont这样的变量中，然后将该变量传入text()，作为最后的关键字参数。❺行的text()调用绘制了Howdy，采用灰色、32点Arial字体。

得到的text.png文件如图17-15所示。

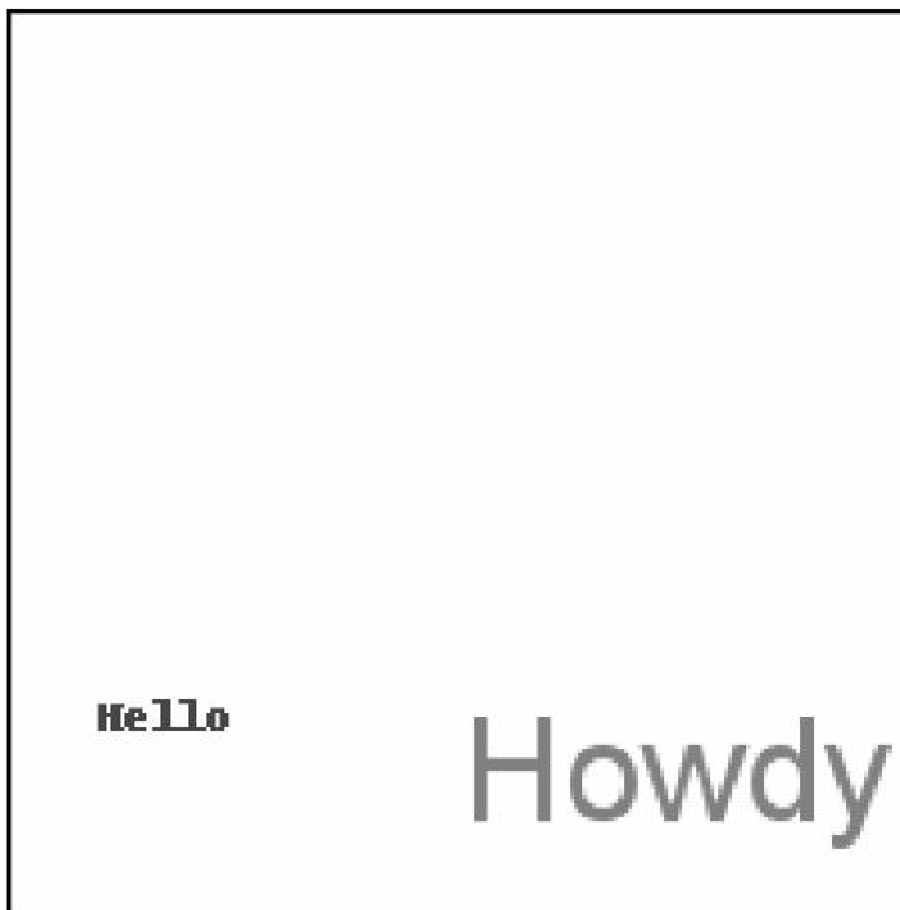


图17-15 得到的图像text.png

17.5 小结

图像由像素的集合构成，每个像素具有表示颜色的RGBA值，可以通过x和y坐标的定位。两种常见的图像格式是JPEG和PNG。Pillow模块可以处理这两种图像格式和其他格式。

当图像被加载为Image对象时，它的宽度和高度作为两整数元组，保存在size属性中。Image数据类型的对象也有一些方法，实现常见的图像处理：crop()、copy()、paste()、resize()、rotate()和transpose()。要将Image对象保存为图像文件，就调用save()方法。

如果希望程序在图像上绘制形状，就使用ImageDraw的方法绘制点、线、矩形、椭圆和多边形。该模块也提供了一些方法，用你选择的字体和大小绘制文本。

虽然像Photoshop这样高级（且昂贵）的应用程序提供了自动批量处理功能，但你可以用Python脚本，免费完成许多相同的修改。在前面的章节中，你编写Python程序来处理纯文本文件、电子表格、PDF和其他格式。利用Pillow模块，你已将编程能力扩展到处理图像！

17.6 习题

1．什么是RGBA值？

2．如何利用Pillow模块得到'CornflowerBlue'的RGBA值？

3．什么是矩形元组？

4．哪个函数针对名为sophie.png图像文件返回一个Image对象？

5．如何得到一个Image对象的图像的宽度和高度？

6．调用什么方法会得到一个100×100的图像的Image对象，但不包括它左下角的四分之一？

7．对Image对象修改后，如何将它保存为图像文件？

8．什么模块包含Pillow的形状绘制代码？

9．Image对象没有绘制方法。哪种对象有？如何获得这种类型的对象？

17.7 实践项目

作为实践，编程完成以下任务。

17.7.1 扩展和修正本章项目的程序

本章的resizeAndAddLogo.py程序使用PNG和JPEG文件，但Pillow还支持许多格式，不仅仅是这两个。扩展resizeAndAddLogo.py，让它也能处理GIF和BMP图像。

另一个小问题是，只有文件扩展名小写时，程序才修改PNG和JPEG文件。例如，它会处理zophie.png，但不处理zophie.PNG。修改代码，让文件扩展名检查不区分大小写。

最后，添加到右下角的徽标本来只是一个小标记，但如果该图像与徽标本身差不多大，结果将类似于图17-16。修改resizeAndAddLogo.py，使得图像必须至少是徽标的两倍的宽度和高度，然后才粘贴徽标。否则，它应该跳过添加徽标。

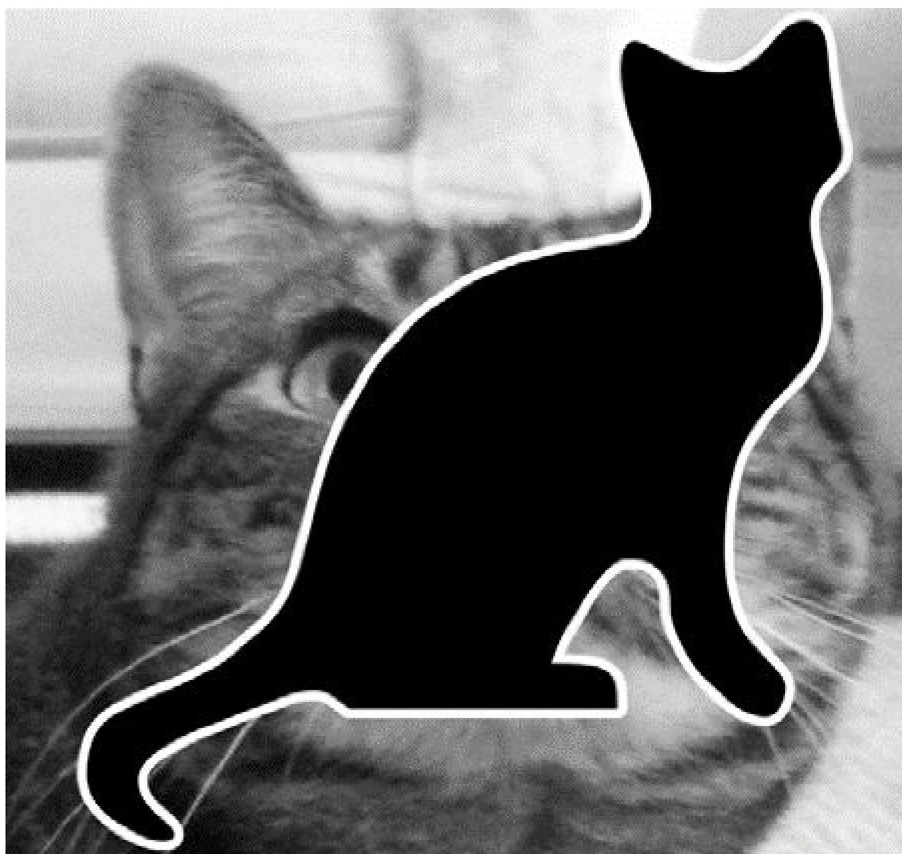


图17-16 如果图像不比徽标大很多，结果很难看。

17.7.2 在硬盘上识别照片文件夹

我有一个坏习惯，从数码相机将文件传输到硬盘的临时文件夹后，会忘记这些文件夹。编程扫描整个硬盘，找到这些遗忘的“照片文件夹”，就太好了。

编写一个程序，遍历硬盘上的每个文件夹，找到可能的照片文件

夹。当然，首先你必须定义什么是“照片文件夹”。假定就是超过半数文件是照片的任何文件夹。你如何定义什么文件是照片？

首先，照片文件必须具有文件扩展名.png或.jpg。此外，照片是很大的图像。照片文件的宽度和高度都必须大于500像素。这是安全的假定，因为大多数数码相机照片，宽度和高度都是几千像素。

作为提示，下面是这个程序的粗略框架：

```
#! python3
# Import modules and write comments to describe this program.

for foldername, subfolders, filenames in os.walk('C:\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Check if file extension isn't .png or .jpg.
        if TODO:
            numNonPhotoFiles += 1
            continue     # skip to next filename

        # Open image file using Pillow.

        # Check if width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            numPhotoFiles += 1
        else:
            # Image is too small to be a photo.
            numNonPhotoFiles += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

程序运行时，它应该在屏幕上打印所有照片文件夹的绝对路径。

17.7.3 定制的座位卡

第13章包含了一个实践项目，利用纯文本文件的客人名单，创建定制的邀请函。作为附加项目，请使用 Pillow 模块，为客人创建定制的座位卡图像。从http://nostarch. com/automatestuff/ 下载资源文件guests.txt，对于其中列出的客人，生成带有客人名字和一些鲜花装饰的图像文件。在http://nostarch.com/automatestuff/ 的资源中，包含一个版权为公共领域的鲜花图像。

为了确保每个座位卡大小相同，在图像的边缘添加一个黑色的矩形，这样在图像打印出来时，可以沿线剪裁。Pillow生成的PNG文件被设置为每英寸72个像素，因此4×5英寸的卡片需要288×360像素的图像。

第18章　用GUI自动化控制键盘和鼠标

知道用于编辑电子表格、下载文件和运行程序的各种Python模块，是很有用的。但有时候，就是没有模块对应你要操作的应用程序。在计算机上自动化任务的终极工具，就是写程序直接控制键盘和鼠标。这些程序可以控制其他应用，向它们发送虚拟的击键和鼠标点击，就像你自己坐在计算机前与应用交互一样。这种技术被称为“图形用户界面自动化”，或简称为“GUI自动化”。有了GUI自动化，你的程序就像一个活人用户坐在计算机前一样，能做任何事情，除了将咖啡泼在键盘上。

请将GUI自动化看成是对一个机械臂编程。你可以对机械臂编程，让它敲键盘或移动鼠标。对于涉及许多无脑点击或填表的任务，这种技术特别有用。

pyautogui模块包含了一些函数，可以模拟鼠标移动、按键和滚动鼠标滚轮。本章只介绍了pyautogui功能的子集。可以在http://pyautogui.readthedocs.org/ 找到完整的文档。

18.1 安装pyautogui模块

pyautogui模块可以向Windows、OS X和Linux发送虚拟按键和鼠标点击。根据你使用的操作系统，在安装pyautogui之前，可能需要安装一些其他模块（称为依赖关系）。

- 在Windows上，不需要安装其他模块。
- 在OS X上，运行sudo pip3 install pyobjc-framework-Quartz，sudo pip3 install pyobjc-core，然后sudo pip3 install pyobjc。
- 在Linux上，运行sudo pip3 install python3-xlib，sudo apt-get install scrot，sudo apt-get install python3-tk，以及sudo apt-get install python3-dev（Scrot是PyAutoGUI使用的屏幕快照程序）。

在这些依赖安装后，运行pip install pyautogui（或在OS X和Linux上运行pip3），安装pyautogui。

附录A有安装第三方模块的完整信息。要测试PyAutoGUI是否正确安装，就在交互式环境运行import pyautogui，并检查出错信息。

18.2 走对路

在开始GUI自动化之前，你应该知道如何避免可能发生的问题。Python能以想象不到的速度移动鼠标并击键。实际上，它可能太快，导致其他程序跟不上。而且，如果出了问题，但你的程序继续到处移动鼠标，可能很难搞清楚程序到底在做什么，或者如何从问题中恢复。就像迪斯尼电影《魔法师的学徒》中的魔法扫帚，它不断地向米老鼠的浴缸注水（然后水溢出来），你的程序可能失去控制，即使它完美地执行你的指令。如果程序自己在移动鼠标，停止它可能很难，你不能点击IDLE窗口来关闭它。好在，有几种方法来防止或恢复GUI自动化问题。

18.2.1 通过注销关闭所有程序

停止失去控制的GUI自动化程序，最简单的方法可能是注销，这将关闭所有运行的程序。在Windows和Linux上，注销的热键是Ctrl-Alt-Del。在OS X，热键是⌘-Shift-Option-Q。通过注销，你会丢失所有未保存的工作，但至少不需要等计算机完全重启。

18.2.2 暂停和自动防故障装置

你可以告诉脚本在每次函数调用后等一会儿，在出问题的时候，让你有很短的时间窗口来控制鼠标和键盘。要做到这一点，将pyautogui.PAUSE变量设置为要暂停的秒数。例如，在设置pyautogui.PAUSE = 1.5之后，每个PyAutoGUI函数调用在执行动作之后，都会等待一秒半。非PyAutoGUI指令不会停顿。

pyautogui 也有自动防故障功能。将鼠标移到屏幕的左上角，这将导致pyautogui产生pyautogui .FailSafeException异常。你的程序可以用try和except语句来处理这个异常，也可以让异常导致程序崩溃。这两种情况下，如果你尽可能快地向左上移动鼠标，自动防故障功能都将停止程序。可以设置pyautogui. FAILSAFE = False，禁止这项功能。在交互式环境中输入以下内容：

```
>>> import pyautogui
```

```
>>> pyautogui.PAUSE = 1

>>> pyautogui.FAILSAFE = True
```

这里我们导入pyautogui，并将pyautogui.PAUSE设置为1，即每次函数调用后暂停一秒。将pyautogui.FAILSAFE设置为True，启动自动防故障功能。

18.3 控制鼠标移动

在本节中，你将学习如何利用pyautogui移动鼠标，追踪它在屏幕上的位置，但首先需要理解pyautogui如何处理坐标。

pyautogui的鼠标函数使用x、y坐标。图18-1中展示了计算机屏幕的坐标系统。它与17章中讨论的图像坐标系统类似。原点的x、y都是零，在屏幕的左上角。向右x坐标增加，向下y坐标增加。所有坐标都是正整数，没有负数坐标。

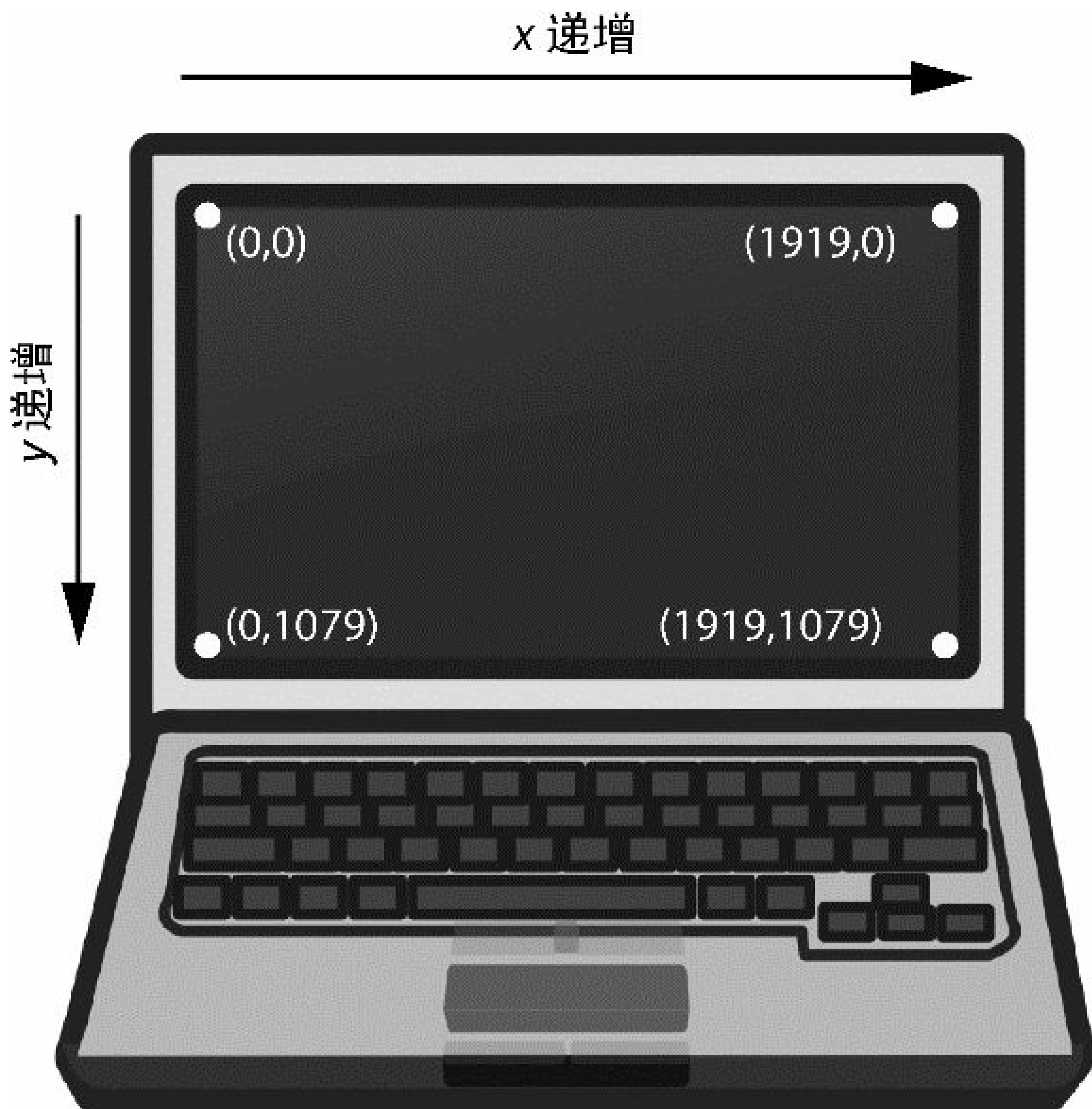


图18-1 分辨率为1920 × 1080的计算机屏幕上的坐标

分辨率是屏幕的宽和高有多少像素。如果屏幕的分辨率设置为1920 × 1080，那么左上角的坐标是（0，0），右下角的坐标是（1919，1079）。

pyautogui.size() 函数返回两个整数的元组，包含屏幕的宽和高的像素数。在交互式环境中输入下面内容：

```
>>> import pyautogui
```

```
>>> pyautogui.size()
(1920, 1080)
>>> width, height = pyautogui.size()
```

在分辨率为1920 × 1080的计算机上，pyautogui.size() 返回（1920，1080）。根据屏幕分辨率的不同，返回值可能不一样。你可以将来自pyautogui.size() 的宽和高存在变量中，如width和height，让程序的可读性更好。

18.3.1 移动鼠标

既然你理解了屏幕坐标，就让我们来移动鼠标。pyautogui.moveTo() 函数将鼠标立即移动到屏幕的指定位置。表示x、y坐标的整数值分别构成了函数的第一个和第二个参数。可选的duration整数或浮点数关键字参数，指定了将鼠标移到目的位置所需的秒数。如果不指定，默认值是零，表示立即移动（在PyAutoGUI函数中，所有的duration关键字参数都是可选的）。在交互式环境中输入以下内容：

```
>>> import pyautogui
>>> for i in range(10):
```

```
    pyautogui.moveTo(100, 100, duration=0.25)

    pyautogui.moveTo(200, 100, duration=0.25)

    pyautogui.moveTo(200, 200, duration=0.25)

    pyautogui.moveTo(100, 200, duration=0.25)
```

这个例子根据提供的坐标，以正方形的模式顺时针移动鼠标，移动了10次。每次移动耗时0.25秒，因为有关键字参数指定 duration=0.25。如果没有指定函数调用的第三个参数，鼠标就会马上从一个点移到另一个点。

pyautogui.moveRel() 函数相对于当前的位置移动鼠标。下面的例子同样以正方形的模式移动鼠标，只是它从代码开始运行时鼠标所在的位置开始，按正方形移动：

```
>>> import pyautogui
```

```
>>> for i in range(10):

        pyautogui.moveRel(100, 0, duration=0.25)

        pyautogui.moveRel(0, 100, duration=0.25)

        pyautogui.moveRel(-100, 0, duration=0.25)

        pyautogui.moveRel(0, -100, duration=0.25)
```

pyautogui.moveRel() 也接受3个参数：向右水平移动多少个像素，向下垂直移动多少个像素，以及（可选的）花多少时间完成移动。为第一第二个参数提供负整数，鼠标将向左或向上移动。

18.3.2 获取鼠标位置

通过调用pyautogui.position() 函数，可以确定鼠标当前的位置。它

将返回函数调用时，鼠标x、y坐标的元组。在交互式环境中输入以下内容，每次调用后请移动鼠标：

```
>>> pyautogui.position()
(311, 622)
>>> pyautogui.position()
(377, 481)
>>> pyautogui.position()
(1536, 637)
```

当然，返回值取决于鼠标的位置。

18.4 项目：“现在鼠标在哪里？”

能够确定鼠标的位置，对于建立GUI自动化脚本是很重要的。但光看屏幕，几乎不能弄清楚像素的准确坐标。如果有一个程序在移动鼠标时随时显示 x y坐标，就会很方便。

总的来说，你希望该程序做到：

- 获得鼠标当前的xy坐标。
- 当鼠标在屏幕上移动时，更新这些坐标。

这意味着代码需要做到下列事情：

- 调用函数取得当前坐标。
- 在屏幕上打印回退制服。删除以前打印的坐标。
- 处理异常。让用户能按键退出。

打开一个新的文件编辑器窗口，将它保存为mouseNow.py。

第1步：导入模块

程序开始是这样的：

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
#TODO: Get and print the mouse coordinates.
```

程序开始导入了pyautogui模块，打印的内容提醒用户按Ctrl-C退出。

第2步：编写退出代码和无限循环

可以用无限 while 循环，不断打印通过mouse.position() 获得的当前鼠标坐标。对于退出程序的代码，你需要捕捉 KeyboardInterrupt 异常，它会在用户按下 Ctrl-C 时抛出。如果不处理这个异常，它会向用户显示丑陋的调用栈和出错信息。将下面内容添加到程序中：

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
try:
```

```
    while True:

        # TODO: Get and print the mouse coordinates.

❶ except KeyboardInterrupt:

❷     print('\nDone.')
```

为了处理这个异常，将无限while循环放在一个try语句中。当用户按下Ctrl-C，程序执行将转到except子句❶，新行中将输出Done❷。

第3步：获取并打印鼠标坐标

while循环内的代码应该获取当前鼠标的坐标，提供好看的格式，并打印输出。在while循环内添加以下代码：

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
import pyautogui
print('Press Ctrl-C to quit.')
--snip
```

```
--
        # Get and print the mouse coordinates.

        x, y = pyautogui.position()

        positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)

--snip

--
```

利用多重赋值的技巧，变量x、y得到了pyautogui.position() 返回元组中的两个整型值。将x、y传递给str() 函数，可以得到整型坐标的字符串形式。rjust() 字符串方法将对坐标右调整，让它们占据同样的宽度，不论坐标是一位、两位、三位或四位数字。连接右调准的字符串坐标，加上 'X: ' 和 ' Y: ' 标签，就得到了格式化好的字符串，保存在positionStr中。

在程序的末尾，添加以下代码：

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip
```

```
--
        print(positionStr, end='')


❶         print('\b' * len(positionStr), end='', flush=True)
```

这将在屏幕上打印positionStr。print() 函数的关键字参数end='' 阻止了在打印行末添加默认的换行字符。这可能会擦除你已经在屏幕上打印的文本，但只是最近一行文本。如果你先打印了一个换行字符，就不会擦除以前打印的内容。

要擦除文本，就打印\b 退格转义字符。这个特殊字符擦除屏幕当前行末尾的字符。代码行❶利用字符串复制，得到了许多\b 字符构成的字符串，长度与positionStr中保存的字符串长度一样，效果就是擦除了前面打印的字符串。

print() 调用打印\b退格键字符时，总是传入flush=True（其技术上的理由超出了本书的范围）。否则，屏幕可能不会按期望更新。

while循环重复非常快，用户实际上不会注意到你在屏幕上删除并重新打印整个数字。例如，如果x坐标是563，鼠标右移一个像素，看起来就像563中的3变成了4。

如果运行程序，只有两行打印输出。看起来像这样：

```
Press Ctrl-C to quit.
X:   290 Y:   424
```

第一行显示指令：按Ctrl-C退出。第二行显示鼠标坐标，当你在屏幕上移动鼠标时，会变化。利用这个程序，就能搞清楚鼠标坐标，用于你的GUI自动化脚本。

18.5 控制鼠标交互

既然你知道了如何移动鼠标，弄清楚了它在屏幕上的位置，就可以开始点击、拖动和滚动鼠标。

18.5.1 点击鼠标

要向计算机发送虚拟的鼠标点击，就调用pyautogui.click() 方法。默认情况下，点击将使用鼠标左键，点击发生在鼠标当前所在位置。如果希望点击在鼠标当前位置以外的地方发生，可以传入x、y坐标作为可选的第一第二参数。

如果想指定鼠标按键，就加入button关键字参数，值分别为'left'、'middle'或 'right'。例如，pyautogui.click（100，150，button='left'）将在坐标（100，150）处点击鼠标左键。而pyautogui.click（200，250，button='right'）将在坐标（200，250）处点击右键。

在交互式环境中输入以下内容：

```
>>> import pyautogui
>>> pyautogui.click(10, 5)
```

你应该看到鼠标移到屏幕左上角的位置，并点击一次。完整的“点击”是指按下鼠标按键，然后放开，同时不移动位置。实现点击也可以调用pyautogui. mouseDown()，这只是按下鼠标按键，再调用pyautogui.mouseUp()，这只是释放鼠标按键。这些函数的参数与click()相同。实际上，click() 函数只是这两个函数调用的方便封装。

为了进一步方便，pyautogui.doubleClick() 函数只执行双击鼠标左键。pyautogui.rightClick() 和pyautogui.middleClick() 函数将分别执行双击右键和双击中键。

18.5.2 拖动鼠标

“拖动”意味着移动鼠标，同时按住一个按键不放。例如，可以通过拖动文件图标，在文件夹之间移动文件，或在日历应用中移动预约。

PyAutoGUI提供了pyautogui.dragTo() 和pyautogui.dragRel() 函数，将鼠标拖动到一个新的位置，或相对当前位置的位置。dragTo() 和dragRel() 的参数与moveTo() 和moveRel() 相同：x坐标/水平移动，y坐标/垂直移动，以及可选的时间间隔（在OS X上，如果鼠标移动太快，拖动会不对，所以建议提供duration关键字参数）。

要尝试这些函数，请打开一个绘图应用，如Windows上的Paint，OS X上的Paintbrush，或Linux上的GNU Paint（如果没有绘图应用，可以使用在线绘图，网址是http://sumopaint.com/ ）。我将使用PyAutoGUI在这些应用中绘图。

让鼠标停留在绘图应用的画布上，同时选中铅笔或画笔工具，在新的文件编辑窗口中输入以下内容，保存为spiralDraw.py：

```
import pyautogui, time
❶ time.sleep(5)
❷ pyautogui.click() # click to put drawing program in focus
distance = 200
while distance > 0:
```

```
❸       pyautogui.dragRel(distance, 0, duration=0.2)  # move right
❹       distance = distance - 5
❺       pyautogui.dragRel(0, distance, duration=0.2)  # move down
❺       pyautogui.dragRel(-distance, 0, duration=0.2) # move left
      distance = distance - 5
      pyautogui.dragRel(0, -distance, duration=0.2) # move up
```

在运行这个程序时，会有5秒钟的延迟❶，让你选中铅笔或画笔工具，并让鼠标停留在画图工具的窗口上。然后spiralDraw.py将控制鼠标，点击画图程序获得焦点❷。如果窗口有闪烁的光标，它就获得了“焦点”，这时你的动作（例如打字，或这个例子中的拖动鼠标），就会影响该窗口。画图程序获取焦点后，spiralDraw.py将绘制一个正方形旋转图案，如图18-2所示。

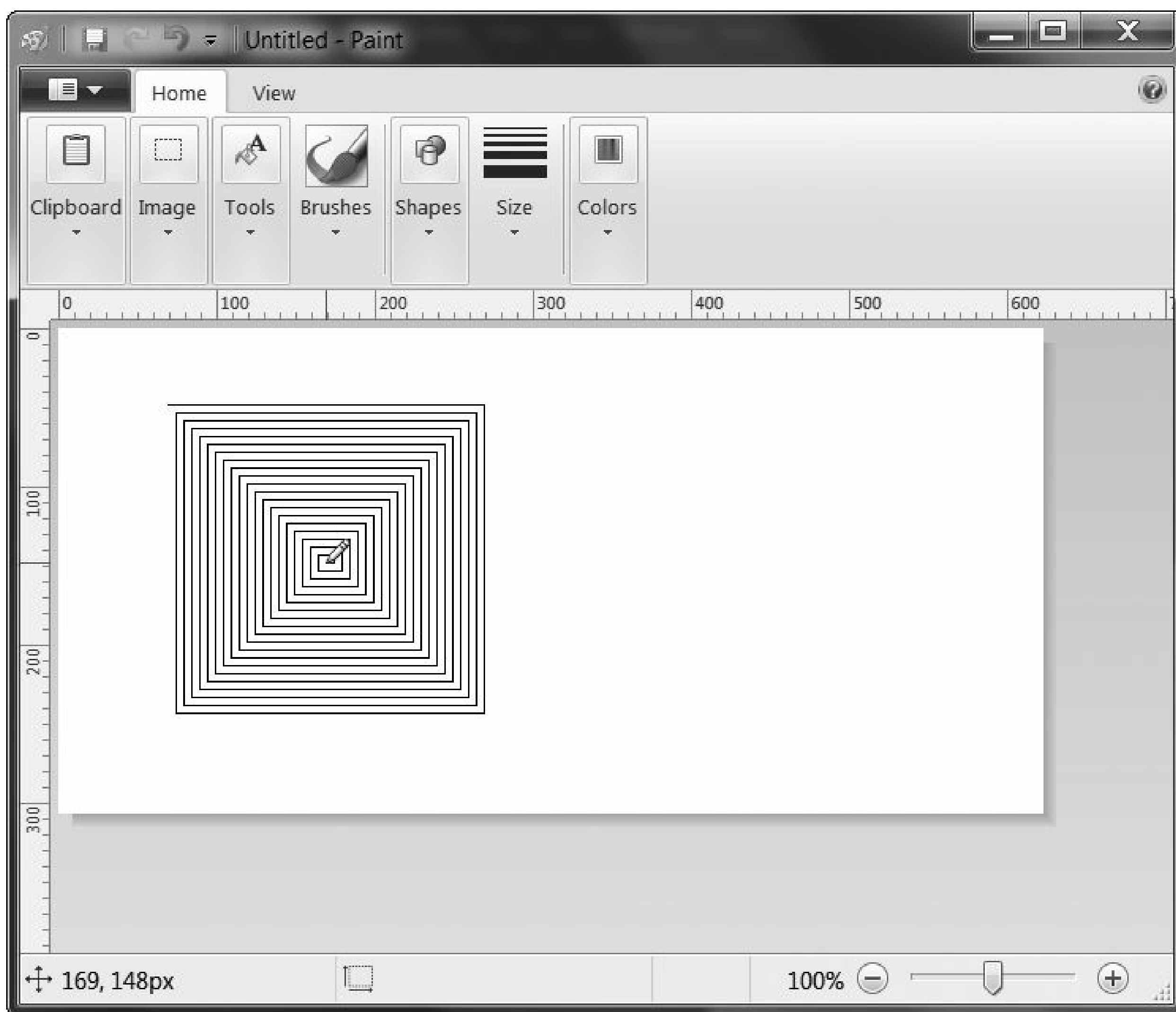


图18-2 pyautogui.dragRel() 例子的结果

distance 变量从 200 开始，所以在 while 循环的第一次迭代中，第一次dragRel() 调用将光标向右拖动200像素，花了0.2秒❸。然后distance降到195❹，第二次dragRel() 调用将光标向下拖动195像素❺。第三次dragRel() 调用将光标水平拖动−195（向左195）❻，distance降到190，最后一次dragRel调用将光标向上拖动190。每次迭代，鼠标都向右、向下、向左、向上拖动，distance都比前一次迭代小一点。通过这段代码循环，就可以移动鼠标光标，画出正方形旋转图案。

可以手工画出这个漩涡（或者说用鼠标），但一定要画得很慢，才能这么精确。pyautogui能够几秒钟就画完。

注意

> 你可以在代码中使用pillow模块的画图函数，画出这个图形，更多信息请参见第17章。但利用GUI自动化就能使用画图程序提供的高级画图工具，如灰度、不同的画笔或填充工具。

18.5.3 滚动鼠标

最后一个pyautogui鼠标函数是scroll()。你向它提供一个整型参数，说明向上或向下滚动多少单位。单位的意义在每个操作系统和应用上不一样，所以你必须试验，看看在你的情况下滚动多远。滚动发生在鼠标的当前位置。传递正整数表示向上滚动，传递负整数表示向下滚动。将鼠标停留在IDLE窗口上，在IDLE的交互式环境中运行以下代码：

```
>>> pyautogui.scroll(200)
```

你会看到IDLE轻松地向上滚动，然后又向下滚回来。发生向下滚动是因为，在执行完指令后，IDLE自动向下滚动到底部。输入以下代码作为替代：

```
>>> import pyperclip
>>> numbers = ''
>>> for i in range(200):
```

```
        numbers = numbers + str(i) + '\n'

>>> pyperclip.copy(numbers)
```

这导入了pyperclip，并建立一个空字符串numbers。代码然后循环200个数字，将每个数字和换行符加入numbers。在pyperclip.copy（numbers）之后，剪贴板中将保存200行数字。打开一个新的文件编辑窗口，将文本粘贴进去。这将得到一个很大的文本窗口，让你尝试滚动。在交互式环境中输入以下代码：

```
>>> import time, pyautogui

>>> time.sleep(5); pyautogui.scroll(100)
```

在第二行，输入的两条命令以分号分隔，这告诉Python在运行这些命令时，就像它们在独立的行中一样。唯一的区别在于，交互式环境不会在两个命令之间提示你输入。这对于这个例子很重要，因为我们希望pyautogui.scroll() 调用在等待之后自动发生（请注意，虽然在交互式环境中，将两条命令放在一行中可能有用，但在你的程序中，还是应该让每条命令独占一行）。

按下回车运行代码后，你有5秒钟的时间点击文件编辑窗口，让它获得焦点。在5秒钟的延迟结束后，pyautogui.scroll() 调用将导致文件编辑窗口向上滚动。

18.6 处理屏幕

你的GUI自动化程序没有必要盲目地点击和输入。pyautogui拥有屏幕快照的功能，可以根据当前屏幕的内容创建图形文件。这些函数也可以返回一个Pillow的Image对象，包含当前屏幕的内容。如果你是跳跃式地阅读本书，可能需要阅读第17章，安装pillow模块，然后再继续本节的内容。

在Linux计算机上，需要安装scrot程序，才能在pyautogui中使用屏幕快照功能。在终端窗口中，执行sudo apt-get install scrot，安装该程序。如果你使用Windows或OS X，就跳过这一步，继续本节的内容。

18.6.1 获取屏幕快照

要在Python中获取屏幕快照，就调用pyautogui.screenshot() 函数。在交互式环境中输入以下内容：

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
```

```
```

im变量将包含一个屏幕快照的Image对象。现在可以调用im变量中Image对象的方法，就像所有其他Image对象一样。在交互式环境中输入以下内容：

```
>>> im.getpixel((0, 0))
(176, 176, 175)
>>> im.getpixel((50, 200))
(130, 135, 144)
```

向getpixel() 函数传入坐标元组，如（0，0）或（50，200），它将告诉你图像中这些坐标处的像素颜色。getpixel() 函数的返回值是一个RGB元组，包含3个整数，表示像素的红绿蓝值（没有第四个值表示alpha，因为屏幕快照是完全不透明的）。这就是你的程序“看到”当前屏幕上内容的方法。

18.6.2 分析屏幕快照

假设你的GUI自动化程序中，有一步是点击灰色按钮。在调用click() 方法之前，你可以获取屏幕快照，查看脚本要点击处的像素。如果它的颜色和灰色按钮不一样，那么程序就知道出问题了。也许窗口发

生了意外的移动，或者弹出式对话框挡住了该按钮。这时，不应该继续（可能会点击到错误的东西，造成严重破坏），程序可以“看到”它没有点击在正确的东西上，并自行停止。

如果屏幕上指定的 x、y 坐标处的像素与指定的颜色匹配，PyAutoGUI 的pixelMatchesColor() 函数将返回True。第一和第二个参数是整数，对应x和y坐标。第三个参数是一个元组，包含3个整数，是屏幕像素必须匹配的RGB颜色。在交互式环境中输入以下内容：

```
  >>> import pyautogui
  >>> im = pyautogui.screenshot()
❶ >>> im.getpixel((50, 200))
  (130, 135, 144)
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
  True
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
  False
```

在获取屏幕快照，并用getpixel() 函数取得特定坐标处像素颜色的RGB元组之后❶，将同样的坐标和RGB元组传递给pixelMatchesColor() ❷，这应该返回True。然后改变RBG元组中的一个值，用同样的坐标再次调用pixelMatches Color() ❸，这应该返回False。你的GUI自动化程序要调用click() 之前，这种方法应该有用。请注意，给定坐标处的颜色应该“完全”匹配。即使只是稍有差异（例如，是（255，255，254）而不是（255，255，255）），那么函数也会返回False。

18.7 项目：扩展**mouseNow**程序

可以扩展本章前面的mouseNow.py项目，让它不仅给出鼠标当前位置的x、 y坐标，也给出这个像素的RGB颜色。将mouseNow.py中while循环内的代码修改为：

```
#! python3
# mouseNow.py - Displays the mouse cursor's current position.
--snip

--
          positionStr = 'X: ' + str(x).rjust(4) + ' Y: ' + str(y).rjust(4)
          pixelColor = pyautogui.screenshot().getpixel((x, y))

          positionStr += ' RGB: (' + str(pixelColor[0]).rjust(3)

          positionStr += ', ' + str(pixelColor[1]).rjust(3)
```

```
        positionStr += ', ' + str(pixelColor[2]).rjust(3) + ')'

        print(positionStr, end='')
--snip
--
```

现在，如果运行mouseNow.py，那么输出将包括鼠标光标处像素的RGB颜色值。

```
Press Ctrl-C to quit.
X:   406 Y:    17 RGB: (161, 50, 50)
```

这个信息，配合pixelMatchesColor() 函数，应该使得给GUI自动化脚本添加颜色检查变得容易。

18.8 图像识别

但是，如果事先不知道应该点击哪里，怎么办？可以使用图像识别。向PyAutoGUI提供希望点击的图像，让它去弄清楚坐标。

例如，如果你以前获得了屏幕快照，截取了提交按钮的图像，保存为submit.png，那么 locateOnScreen() 函数将返回图像所在处的坐标。要了解 locateOnScreen()函数的工作方式，请获取屏幕上一小块区域的屏幕快照，保存该图像，并在交互式环境中输入以下内容，用你的屏幕快照文件名代替 'submit. png'：

```
>>> import pyautogui
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
```

locateOnScreen() 函数返回 4 个整数的元组，是屏幕上首次发现该图像时左边的x坐标、顶边的y坐标、宽度以及高度。如果你用自己的屏幕快照，在你的计算机上尝试，那么返回值会和这里显示的不一样。

如果屏幕上找不到该图像，locateOnScreen() 函数将返回None。请注意要成功识别，屏幕上的图像必须与提供的图像完全匹配。即使只差一个像素，locateOn Screen() 函数也会返回None。

如果该图像在屏幕上能够找到多处，locateAllOnScreen() 函数将返回一个Generator对象。可以将它传递给list()，返回一个4整数元组的列表。继续在交互式环境的例子中输入以下内容（用你自己的图像文件名取代 'submit.png'）：

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

每个 4 整数元组代表了屏幕上的一个区域。如果图像只找到一次，返回的列表就只包含一个元组。

在得到图像所在屏幕区域的4整数元组后，就可以点击这个区域的中心。将元组传递给center() 函数，它将返回该区域中心的x、y坐标。在交互式环境中输入以下内容，用你自己的文件名、4整数元组和坐标对，来取代参数：

```
>>> pyautogui.locateOnScreen('submit.png')
(643, 745, 70, 29)
>>> pyautogui.center((643, 745, 70, 29))
(678, 759)
>>> pyautogui.click((678, 759))
```

用center() 得到中心坐标后，将click() 坐标传递给函数，就会点击屏幕上该区域的中心，这个区域匹配你传递给locateOnScreen() 函数的图像。

18.9 控制键盘

pyautogui也有一些函数向计算机发送虚拟按键，让你能够填充表格，或在应用中输入文本。

18.9.1 通过键盘发送一个字符串

pyautogui.typewrite() 函数向计算机发送虚拟按键。这些按键产生什么效果，取决于当前获得焦点的窗口和文本输入框。可能需要先向文本输入框发送一次鼠标点击，确保它获得焦点。

举一个简单的例子，让我们用Python自动化在文件编辑窗口中输入Hello world!。首先，打开一个新的文件编辑窗口，将它放在屏幕的左上角，以便pyautogui点击正确的位置，让它获得焦点。然后，在交互式环境中输入以下内容：

```
>>> pyautogui.click(100, 100); pyautogui.typewrite('Hello world!')
```

请注意，在同一行中放两条命令，用分号隔开，这让交互式环境不会在两个指令之间提示输入。这防止了你在click() 和typewrite() 调用之间，不小心让新的窗口获得焦点，从而让这个例子失败。

Python首先在坐标（100，100）处发出虚拟鼠标点击，这将点击文件编辑窗口，让它获得焦点。typewrite() 函数调用将向窗口发送文本Hello world!，结果就像图18-3。现在有了替你打字的代码！

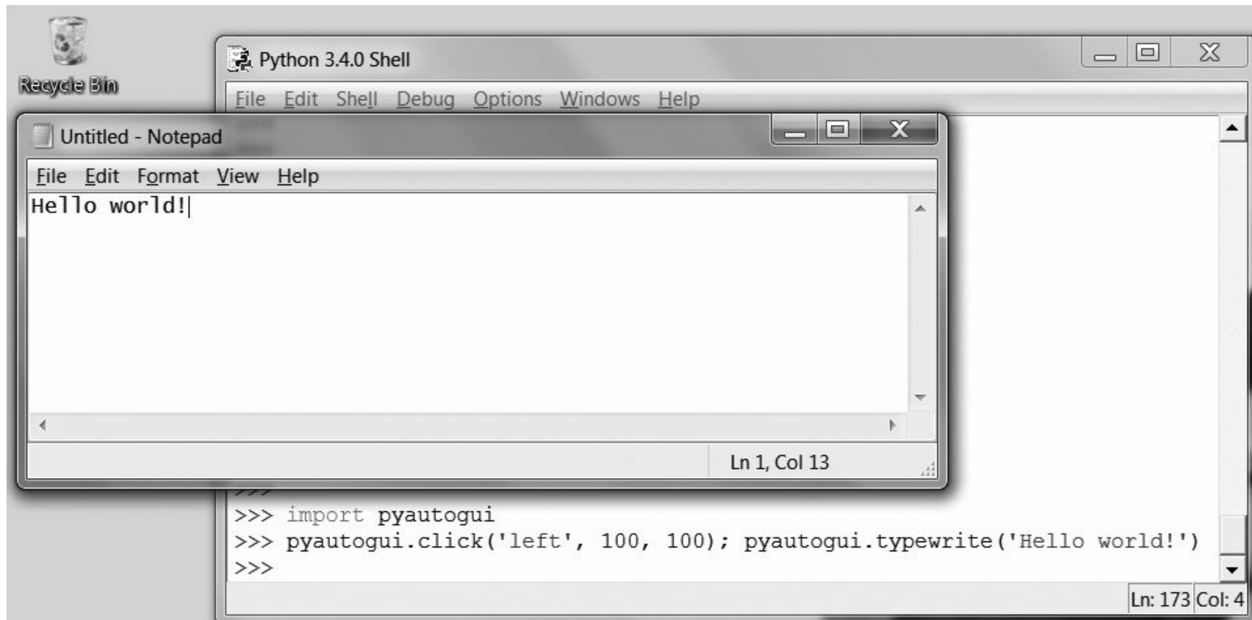


图18-3 用PyAutogGUI点击文件编辑器窗口，在其中输入Hello world！

默认情况下，typewrite() 函数将立即打印出完整字符串。但是，你可以传入可选的第二参数，在每个字符之间添加短时间暂停。例如，pyautogui.typewrite（'Helloworld!'，0.25）将在打出H后等待1/4秒。打出e以后再等待1/4秒，如此等等。这种渐进的打字机效果，对于较慢的应用可能有用，它们处理击键的速度不够快，跟不上pyautogui。

对于A或!这样的字符，pyautogui将自动模拟按住Shift键。

18.9.2 键名

不是所有的键都很容易用单个文本字符来表示。例如，如何把Shift键或左箭头键表示为单个字符？在PyAutoGUI中，这些键表示为短的字符串值：'esc' 表示Esc键，'enter' 表示Enter。

除了单个字符串参数，还可以向typewrite() 函数传递这些键字符串的列表。例如，以下的调用表示按a键，然后是b键，然后是左箭头两次，最后是X和Y键：

```
>>> pyautogui.typewrite(['a', 'b', 'left', 'left', 'X', 'Y'])
```

因为按下左箭头将移动键盘光标，所以这会输出XYab。表18-1列出了pyautogui的键盘键字符串，你可以将它们传递给typewrite() 函数，模拟任何按键组合。

也可以查看pyautogui.KEYBOARD_KEYS列表，看看pyautogui接受的所有可能的键字符串。'shift' 字符串指的是左边的Shift键，它等价于 'shiftleft'。'ctrl'、'alt' 和 'win' 字符串也一样，它们都是指左边的键。

表**18-1 PyKeyboard**属性

键盘键字符串	含义
'a', 'b', 'c', 'A', 'B', 'C', '1', '2', '3', '!', '@', '#'，等等	单个字符的键
'enter'（or 'return' or '\n'）	回车键
'esc'	Esc键
'shiftleft', 'shiftright'	左右Shift键
'altleft', 'altright'	左右Alt键
'ctrlleft', 'ctrlright'	左右Ctrl键
'tab'（or '\t'）	Tab键
'backspace', 'delete'	Backspace和Delete键

'pageup', 'pagedown'	Page Up和Page Down键
'home', 'end'	Home和End键
'up', 'down', 'left', 'right'	上下左右箭头键
'f1', 'f2', 'f3'，等等	F1至F12键
'volumemute', 'volumedown', 'volumeup'	静音、减小音量、放大音量键（有些键盘没有这些键，但你的操作系统仍能理解这些模拟的按键）
'pause'	Pause键
'capslock', 'numlock', 'scrolllock'	Caps Lock，Num Lock和Scroll Lock键
'insert'	Ins或Insert键
'printscreen'	Prtsc或Print Screen键
'winleft', 'winright'	左右Win键（在Windows上）
'command'	Command键（在OS X上）
'option'	Option键（在OS X上）

18.9.3 按下和释放键盘

就像mouseDown() 和mouseUp() 函数一样，pyautogui.keyDown() 和pyautogui. keyUp() 将向计算发送虚拟的按键和释放。它们将根据参数发送键字符串（参见表18-1）。方便起见，pyautogui提供了

pyautogui.press() 函数，它调用这两个函数，模拟完整的击键。

运行下面的代码，它将打印出美元字符（通过按住Shift键并按4得到）：

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shif
```

这行代码按下Shift，按下（并释放）4，然后再释放Shift。如果你需要在文本输入框内打一个字符串，typewrite() 函数就更适合。但对于接受单个按键命令的应用，press() 函数是更简单的方式。

18.9.4 热键组合

“热键”或“快捷键”是一种按键组合，它调用某种应用功能。拷贝选择内容的常用热键是Ctrl-C（在Windows和Linux上）或⌘-C（在OS X上）。用户按住Ctrl键，然后按C键，然后释放C和Ctrl键。要用pyautogui的keyDown() 和keyUp() 函数来做到这一点，必须输入以下代码：

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

这相当复杂。作为替代，可以使用pyautogui.hotkey() 函数，它接受多个键字符串参数，按顺序按下，再按相反的顺序释放。例如对于Ctrl-C，代码就像下面这样简单：

```
pyautogui.hotkey('ctrl', 'c')
```

对于更大的热键组合，这个函数特别有用。在Word中，Ctrl-Alt-Shift-S热键组合显示Style（样式）窗口。不必使用8次不同的函数调用（4次keyDown() 调用和4次keyUp() 调用），你只要调用hotkey（'ctrl', 'alt', 'shift', 's'）。

在屏幕的左上角打开一个新的IDLE文件编辑窗口，在交互式环境中输入以下内容（在OS X中，用 'ctrl' 代替 'alt'）：

```
  >>> import pyautogui, time

  >>> def commentAfterDelay():

❶         pyautogui.click(100, 100)

❷         pyautogui.typewrite('In IDLE, Alt-3 comments out a line.')

          time.sleep(2)

❸         pyautogui.hotkey('alt', '3')
```

```
>>> commentAfterDelay()
```

这定义了一个函数commentAfterDelay()，在被调用时，将点击文件编辑窗口，让它获得焦点❶，输出“In IDLE，Alt-3 comments out a line”❷，暂停 2 秒钟，然后模拟按下Alt-3热键（或OS X上的ctrl-3）❸。这个快捷键在当前行加上两个#字符，将它注释掉（在IDLE中编写你自己的代码时，这是一个有用的技巧，应该知道）。

18.10 复习PyAutoGUI的函数

本章介绍了许多不同函数，下面是快速的汇总参考：

moveTo（x，y）将鼠标移动到指定的x、y坐标。

moveRel（xOffset，yOffset）相对于当前位置移动鼠标。

dragTo（x，y）按下左键移动鼠标。

dragRel（xOffset，yOffset）按下左键，相对于当前位置移动鼠标。

click（x，y，button）模拟点击（默认是左键）。

rightClick() 模拟右键点击。

middleClick() 模拟中键点击。

doubleClick() 模拟左键双击。

mouseDown（x，y，button）模拟在x、y处按下指定鼠标按键。

mouseUp（x，y，button）模拟在x、y处释放指定键。

scroll（units）模拟滚动滚轮。正参数表示向上滚动，负参数表示向下滚动。

typewrite（message）键入给定消息字符串中的字符。

typewrite（[key1，key2，key3]）键入给定键字符串。

press（key）按下并释放给定键。

keyDown（key）模拟按下给定键。

keyUp（key）模拟释放给定键。

hotkey（[key1，key2，key3]）模拟按顺序按下给定键字符串，然后以相反的顺序释放。

screenshot() 返回屏幕快照的Image对象（参见第17章关于Image对象的信息）。

18.11 项目：自动填表程序

在所有无聊的任务中，填表是最烦人的。到了现在，在最后一章的项目中，你将搞定它。假设你在电子表格中有大量的数据，必须重复将它输入到另一个应用的表单界面中，没有实习生帮你完成。尽管有些应用有导入功能，让你上传包含信息的电子表格，但有时候似乎没有其他方法，只好不动脑子地点击和输入几个小时。读到了本书的这一章，你“当然”知道会有其他方法。

本项目的表单是 Google Docs 表单，你可以在 http://nostarch.com/automatestuff 找到，如图18-4所示。

图18-4 本项目用到的表单

总的来说，你的程序应该做到：

- 点击表单的第一个文本字段。
- 遍历表单，在每个输入栏键入信息。
- 点击Submit按钮。
- 用下一组数据重复这个过程。

这意味着代码需要做下列事情：

- 调用pyautogui.click() 函数，点击表单和Submit按钮。
- 调用pyautogui.typewrite() 函数，在输入栏输入文本。
- 处理KeyboardInterrupt异常，这样用户能按Ctrl-C键退出。

打开一个新的文件编辑器窗口，将它保存为formFiller.py。

第**1**步：弄清楚步骤

在编写代码之前，你需要弄清楚填写一次表格时，需要的准确击键和鼠标点击。18.4节中的mouseNow.py脚本可以帮助你弄清楚确切的鼠标坐标。你只需要知道第一个文本输入栏的坐标。在点击第一个输入栏之后，你可以Tab键，将焦点移到下一个输入栏。这让你不必弄清楚每一个输入栏的x、y坐标。

下面是在表单中输入数据的步骤：

1．点击Name输入栏（在将浏览器窗口最大化后，用mouseNow.py程序来确定坐标。在OS X上，可能需要点击两次：一次让浏览器获得焦点，第二次让Name输入栏获得焦点）。

2．键入名称，然后按Tab键。

3．键入最大的恐惧（greatest fear），然后按Tab键。

4．按向下键适当的次数，选择魔力源（wizard power source）：一次是Wand，两次是Amulet，三次是Crystal ball，四次是money。然后按Tab键（请注意，在OS X中，你必须为每次选择多按一次向下键。对于某些浏览器，你也需要按回车键）。

5．按向右键，选择RoboCop问题的答案。按一次是2，两次是3，三次是4，四次是5，或按空格键选择1（它是默认加亮的）。然后按Tab键。

6．键入附加的备注，然后按Tab键。

7．按回车键，点击“Submit”按钮。

8．在提交表单后，浏览器将转到一个页面。然后你需要点击一个

链接，返回到表单页面。

请注意，如果你稍后再次运行这个程序，可能需要更新鼠标点击的坐标，因为浏览器窗口可能已经改变了位置。要避免这一点，请一直确保浏览器窗口最大化，然后再寻找第一个表单输入框的坐标。而且，不同操作系统上的不同浏览器，工作起来可能与这里的步骤稍有不同，所以在运行程序之前，要确保这些击键组合适合你的计算机。

第2步：建立坐标

在浏览器中载入示例表单（图18-4），并将浏览器窗口最大化。打开一个新的终端窗口或命令行窗口，来运行mouseNow.py脚本，然后将鼠标放在输入框上，弄清楚它的x、y坐标。这些数字将赋给程序中的变量。同时，找出蓝色Submit按钮的x、y坐标和RBG值。这些值将分别赋给变量submitButton和submitButtonColor。

接下来，在表单中填入一些假的数据，点击Submit。你需要看到下一个页面的样子，以便使用程序mouseNow.py寻找这个页面中Submit another response链接的坐标。

让你的源代码看起来像下面的样子。确保用自己测试得到的坐标代替斜体的值：

```
#! python3
# formFiller.py - Automatically fills in the form.

import pyautogui, time

# Set these to the correct coordinates for your computer.
nameField = (648, 319)
submitButton = (651
```

```
, 817

)
  submitButtonColor = (75

, 141

, 249

)
  submitAnotherLink = (760

, 224

)
  # TODO: Give the user a chance to kill the script.

  # TODO: Wait until the form page has loaded.

  # TODO: Fill out the Name Field.

  # TODO: Fill out the Greatest Fear(s) field.

  # TODO: Fill out the Source of Wizard Powers field.
```

```
# TODO: Fill out the RoboCop field.

# TODO: Fill out the Additional Comments field.

# TODO: Click Submit.

# TODO: Wait until form page has loaded.

# TODO: Click the Submit another response link.
```

现在你需要实际想要输入这张表格的数据。在真实世界中，这些数据可能来自电子表格、纯文本文件或某个网站。可能需要额外的代码，将数据加载到程序中。但对于这个项目，只需要将这些数据硬编码给一个变量。在程序中加入以下代码：

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip

--

formData = [{'name': 'Alice', 'fear': 'eavesdroppers', 'source': 'wand',

 'robocop': 4, 'comments': 'Tell Bob I said hi.'},
```

```
{'name': 'Bob', 'fear': 'bees', 'source': 'amulet', 'robocop': 4,

'comments': 'n/a'},

{'name': 'Carol', 'fear': 'puppets', 'source': 'crystal ball',

'robocop': 1, 'comments': 'Please take the puppets out of the
```

```
break room.'},

{'name': 'Alex Murphy', 'fear': 'ED-209', 'source': 'money',

'robocop': 5, 'comments': 'Protect the innocent. Serve the public

trust. Uphold the law.'},
```

```
]

--snip

--
```

formData列表包含4个字典，针对4个不同的名字。每个字典都有文本字段的名字作为键，响应作为值。最后一点准备是设置pyautogui的PAUSE变量，在每次函数调用后等待半秒钟。在程序的formData赋值语句后，添加下面的代码：

pyautogui.PAUSE = 0.5

第**3**步：开始键入数据

for循环将迭代formData列表中的每个字典，将字典中的值传递给pyautogui函数，最后在文本输入区输入。

在程序中添加以下代码：

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip
```

```
--
  for person in formData:

 # Give the user a chance to kill the script.

 print('>>> 5 SECOND PAUSE TO LET USER PRESS CTRL-C
<<<')
❶      time.sleep(5)
```

```
# Wait until the form page has loaded.

❷      while not pyautogui.pixelMatchesColor(submitButton[0], submitButton[

submitButtonColor):

time.sleep(0.5)

 --snip

--
```

作为一个小的安全功能，该脚本有 5 秒暂停❶。如果发现程序在做一些预期之外的事，这让用户有机会按 Ctrl-C（或将鼠标移到屏幕的左上角，触发FailSafeException异常），从而关闭程序。然后程序等待，直到Submit按钮的颜色可见❷，这让程序知道，表单页面已经加载了。回忆一下，你在第2步中已经弄清楚了坐标和颜色信息，并将它们保存在submitButton和submitButtonColor变量中。要调用pixelMatchesColor()，就传递坐标submitButton[0] 和submitButton[1]，以及颜色submitButtonColor。

在等待Submit按钮颜色可见的代码之后，添加以下代码：

```
  #! python3
  # formFiller.py - Automatically fills in the form.

-  -snip

--

❶      print('Entering %s info...' % (person['name']))

❷      pyautogui.click(nameField[0], nameField[1])
```

```
 # Fill out the Name field.
❸      pyautogui.typewrite(person['name'] + '\t')

 # Fill out the Greatest Fear(s) field.
❹      pyautogui.typewrite(person['fear'] + '\t')

  --snip
--
```

我们添加了偶尔的print() 调用，在终端窗口中显示程序的状态，让用户知道进展。❶

既然程序知道表格已经加载，就可以调用click()，点击Name输入框

❷，并调用typewrite()，输入person['name'] 中的字符串❸。字符串末尾加上了 '\t' 字符，模拟按下Tab键，它将输入焦点转向下一个输入框，Greatest Fear（s）。另一次typewrite() 调用，将在这个输入框中输入person['fear'] 中的字符串，然后用Tab跳到表格的下一个输入框❹。

第4步：处理选择列表和单选按钮

“wizard powers”问题的下拉菜单和RoboCop字段的单选按钮，处理起来比文本输入框需要更多技巧。要用鼠标点选这些选项，你必须搞清楚每个可能选项的x、y坐标。然而，用箭头键来选择会比较容易。

在程序中加入以下代码：

```
#! python3
# formFiller.py - Automatically fills in the form.

--snip

--

# Fill out the Source of Wizard Powers field.

❶     if person['source'] == 'wand':

❷         pyautogui.typewrite(['down', '\t'])
```

```
elif person['source'] == 'amulet':

pyautogui.typewrite(['down', 'down', '\t'])

elif person['source'] == 'crystal ball':

pyautogui.typewrite(['down', 'down', 'down', '\t'])
```

```
elif person['source'] == 'money':

pyautogui.typewrite(['down', 'down', 'down', 'down', '\t'])

# Fill out the RoboCop field.

❸      if person['robocop'] == 1:

❹           pyautogui.typewrite([' ', '\t'])
```

```
elif person['robocop'] == 2:

pyautogui.typewrite(['right', '\t'])

elif person['robocop'] == 3:

pyautogui.typewrite(['right', 'right', '\t'])
```

```
elif person['robocop'] == 4:

pyautogui.typewrite(['right', 'right', 'right', '\t'])

elif person['robocop'] == 5:

pyautogui.typewrite(['right', 'right', 'right', 'right', '\t'])

 --snip
```

```
--
```

在下拉菜单获得焦点后（回忆一下，你写了代码，在填充Greatest Fear（s）输入框后模拟了按 Tab 键），按下向下箭头，就会移动到选择列表的下一项。根据person['source'] 中的值，你的程序应该发出几次向下按键，然后再切换到下一个输入区。如果这个用户词典中的 'source' 值是 'wand' ❶，我们模拟按向下键一次（选择Wand），并按Tab键❷。如果 'source' 键的值是 'amulet'，模拟按向下键两次，并按Tab键。对其他可能的值也是类似。

RoboCop问题的单选按钮，可以用向右键来选择。或者，如果你想选择第一个选项❸，就按空格键❹。

第5步：提交表单并等待

可以用函数typewrite() 填写备注输入框，将person['comments'] 作为参数。你可以另外输入 '\t'，将焦点移到下一个输入框或Submit按钮。当Submit按钮获得焦点后，调用pyautogui.press（'enter'），模拟按下回车键，提交表单。在提交表单之后，程序将等待5秒，等下一页加载。

在新页面加载之后，它会有一个Submit another response链接，让浏览器转向一个新的、全空的表单页面。在第二步，你已将这个链接的坐标作为元组保存在submitAnotherLink中，所以将这些坐标传递给pyautogui.click()，点击这个链接。

新的表单准备好后，脚本的外层for循环将继续下一次迭代，在表单中输入下一个人的信息。

添加以下代码，完成你的程序：

```
#! python3
# formFiller.py - Automatically fills in the form.
```

```
--snip--

 # Fill out the Additional Comments field.

 pyautogui.typewrite(person['comments'] + '\t')

 # Click Submit.
```

```
pyautogui.press('enter')

# Wait until form page has loaded.

print('Clicked Submit.')

time.sleep(5)
```

```
# Click the Submit another response link.

pyautogui.click(submitAnotherLink[0], submitAnotherLink[1])
```

在主for循环完成后，程序应该已经插入了每个人的信息。在这个例子中，只有4个人要输入。但如果有4000个人，那么编程来完成这个任务将节省大量的输入时间。

18.12 小结

用pyautogui模块实现GUI自动化，通过控制键盘和鼠标，让你与计算机上的应用程序交互。虽然这种方式相当灵活，可以做任何人类用户做的事情，但也有不足之处，即这些程序对它们的点击和键入是相当盲目的。在编写GUI自动化程序时，请试着确保它们在得到错误指令时快

速崩溃。崩溃很烦人，但比程序继续错误要好得多。

利用pyautogui，你可以在屏幕上移动鼠标，模拟鼠标点击、击键和快捷键。pyautogui模块也能检查屏幕上的颜色，让GUI自动化程序对屏幕内容有足够的了解，知道它是否有偏差。甚至可以向它提供一个屏幕快照，让它找出你希望点击的区域坐标。

可以组合使用所有这些pyautogui功能，在计算机上自动化各种无脑的重复任务。实际上，看着鼠标自己移动，看着文本自动出现在屏幕上，这是彻头彻尾的催眠。为什么不用节省下来的时间，舒舒服服地坐着，看着程序为你工作？看着你的聪明才智帮你省去无聊的工作，肯定会让你感到满意。

18.13 习题

1．如何触发pyautogui的失效保护来停止程序？

2．什么函数返回当前的分辨率？

3．什么函数返回鼠标当前位置的坐标？

4．pyautogui.moveTo() 和pyautogui.moveRel() 函数之间的区别是什么？

5．什么函数用于拖放鼠标？

6．调用什么函数将替你键入字符串"Hello world!"？

7．如何模拟按下向左键这样的特殊键？

8．如何将当前屏幕的内容保存为图形文件并命名为screenshot.png？

9．什么代码能够设置每次pyautogui函数调用后暂停两秒钟？

18.14 实践项目

作为实践，编程完成下面的内容。

18.14.1 看起来很忙

许多即时通信程序通过一段时间鼠标不动（例如10分钟），来判断你空闲或离开了计算机。也许你想从桌子边溜走一段时间，但不想让别人看到你的即时通信软件转为空闲状态。请编写一段脚本，每隔10秒钟稍微动一下鼠标。这种移动应该相当小，以便在脚本运行时，如果你需要使用计算机，它也不会给你制造麻烦。

18.14.2 即时通信机器人

Google Talk、Skype、Yahoo Messenger、AIM和其它即时通信应用通常使用专有协议，让其他人很难通过编写Python模块与这些程序交互。但即使这些专有协议，也不能阻止你编写GUI自动化工具。

Google Talk应用有一个搜索条，让你在输入朋友列表中的用户名并按下回车时，打开一个消息窗口。键盘焦点自动移到那个新的窗口。其他即时通信应用也有类似的方式，来打开新的消息窗口。请编写一个应用程序，向朋友列表中选定的一组人发出一条通知消息。程序应该能够处理异常情况，比如朋友离线，聊天窗口出现在屏幕上不同的位置，或确认对话框打断输入消息。程序必须使用屏幕快照，指导它的GUI交互，并在虚拟按键发送之前采用各种检测方式。

> 注意
>
> 你可能需要建立一些假的测试账户，这样就不会在编写这个程序时，不小心打扰真正的朋友。

18.14.3 玩游戏机器人指南

有一个很不错的指南名为“How to Build a Python Bot That Can Play Web Games”，网址是http://nostarch.com/automatestuff/。这份指南解释了如何用Python创建一个GUI自动化程序，玩一个名为Sushi Go Round的Flash游戏。这个游戏需要点击正确的成分按钮，填写客户的寿司订单。填写无错订单越快，得分就越高。这个任务特别适合GUI自动化程序，因为可以作弊得到高分！这份指南包含了本章介绍的许多主题，也涉及PyAutoGUI的基本图像识别功能。

附录A　安装第三方模块

除了Python自带的标准库，其他开发者写了一些自己的模块，进一步扩展了Python的功能。安装第三方模块的主要方法是使用Python的pip工具。这个工具从Python软件基金会的网站https://pypi.python.org/ 安全地下载Python模块，并安装到您的计算机上。PyPI或Python包索引，就像是Python模块的免费应用程序商店。

A.1 pip工具

pip工具的可执行文件在Windows上称为pip，在OS X和Linux上称为pip3。在Windows上，pip位于C:\Python34\Scripts\pip.exe。在OS X上，它位于/Library/ Frameworks/Python.framework/Versions/3.4/bin/pip3。在Linux上，它位于/usr/bin/pip3。

虽然在Windows和OS X上pip会随Python3.4自动安装，但在Linux上，必须单独安装。要在Ubuntu或Debian Linux上安装pip3，就打开一个新的终端窗口，输入get install python3-pip。要在Fedora Linux上安装pip3，就在终端窗口输入install python3 -pip。为了安装这个软件，需要输入计算机的管理员密码。

A.2 安装第三方模块

pip工具需要在命令行中运行：向它传入install命令，跟上想要安装的模块名称。例如，在Windows上，会输入pip install ModuleName，其中ModuleName是模块的名称。在OS X和Linux，必须加sudo前缀来运行pip3，授予管理权限来安装该模块。需要输入sudo pip3 install ModuleName。

如果你已经安装了模块，但想升级到PyPI上提供的最新版本，就运行pip install –U ModuleName（或在OS X和Linux上运行pip3 install –U ModuleName）。

安装模块后，可以在交互式环境中运行import ModuleName，测试

安装是否成功。如果未显示错误信息，就可以认为该模块已经成功安装。

运行下面列出的命令，你可以安装本书中介绍的所有模块（请记住，如果在OS X或Linux上，用pip3替代pip）。

- pip install send2trash
- pip install requests
- pip install beautifulsoup4
- pip install selenium
- pip install openpyxl
- pip install PyPDF2
- pip install python-docx（安装python-docx，而不是docx）
- pip install imapclient
- pip install pyzmail
- pip install twilio
- pip install pillow
- pip install pyobjc-core（仅在OS X上）
- pip install pyobjc（仅在OS X上）
- pip install python3-xlib（仅在Linux上）
- pip install pyautogui

> 注意
>
> 对于OS X用户：pyobjc模块需要20分钟或更长的时间来安装，因此，如果它需要一段时间，不要惊慌。也应该先安装pyobjc核心模块，这将减少整体安装时间。

附录B　运行程序

如果你在IDLE的文件编辑器中打开了一个程序，运行它很简单，按F5或选择Run►Run Module菜单项。这是在编程时运行程序的最简单方法，但打开IDLE来运行已完成的程序可能有点累。执行Python脚本还有更方便的方法。

B.1 第一行

所有Python程序的第一行应该是#!行，它告诉计算机你想让Python来执行这个程序。该行以#!开始，但剩下的内容取决于操作系统。

- 在Windows上，第一行是 #! python3。
- 在OS X，第一行是 #! /usr/bin/env python3。
- 在Linux上，第一行是 #! /usr/bin/python3。

没有#!行，你也能从IDLE运行Python脚本，但从命令行运行它们就需要这一行。

B.2 在Windows上运行Python程序

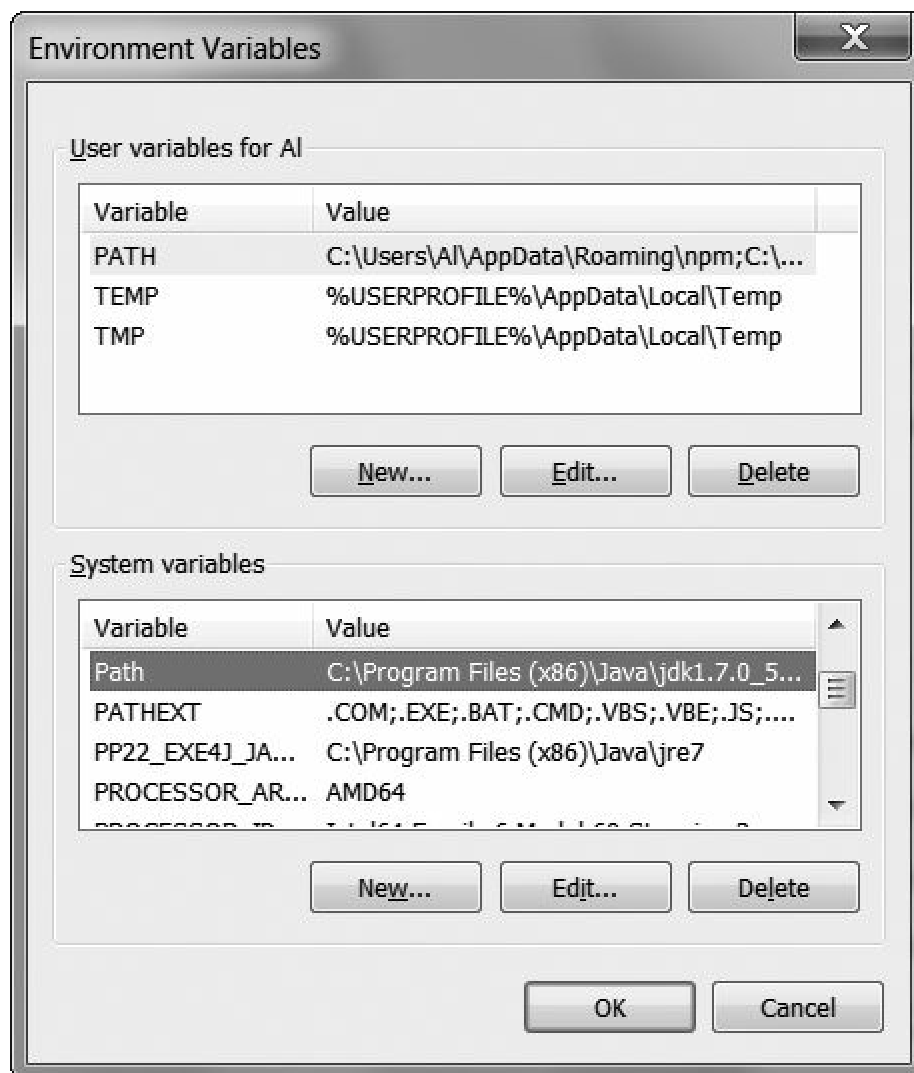
在Windows上，Python3.4的解释程序位于C:\Python34\python.exe。或者，方便的py.exe程序将读取.py文件源代码顶部的#!行，并针对该脚本运行相应的Python版本。如果计算机上安装了多个版本的Python，py.exe程序确保运行正确版本的Python程序。

为了方便运行你的Python程序，可以创建一个.BAT批处理文件，用py.exe来运行Python程序。要创建一个批处理文件，就创建一个新的文本文件，包含一行内容，类似下面这样：

```
@py.exe C:\path\to\your\pythonScript.py %*
```

用你自己的程序的绝对路径替换该路径，将这个文件以.bat 文件扩展名保存（例如，pythonScript.bat）。这个处理文件将使你不必在每次运行时，都输入Python程序完整的绝对路径。我建议将所有的批处理文件和.py 文件放在一个文件夹中，如C:\MyPythonScripts或C:\Users\YourName\PythonScripts。

在Windows上，C:\MyPythonScripts文件夹应该添加到系统路径中，这样就可以从Run对话框中运行其中的批处理文件。要做到这一点，请修改PATH环境变量。单击“开始”按钮，并输入“Edit environment variables for your account（编辑账户的环境变量）”。在你开始输入时，该选项应自动完成。弹出的环境变量窗口如图B-1所示。



图B-1 Windows的环境变量窗口

从系统变量中，选择Path变量，然后单击“编辑”。在“变量值”文本字段中，追加一个分号，键入C:\MyPythonScripts，然后单击“确定”。现在你只需按下Win-R并输入脚本的名称，就能运行C:\MyPythonScripts文件夹中的Python脚本。例如，运行pythonScript，将运行pythonScript.bat，这使你不必从Run对话框运行整个命令py.exe C:\MyPythonScripts\pythonScript.py。

B.3 在OS X和Linux上运行Python程序

在OS X上，选择Applications►Utilities►Terminal将弹出一个终端窗口。终端窗口让你用纯文本在计算机上输入命令，而不是通过图形界

面点击。要在Ubuntu Linux上打开终端窗口，就按Win（或Super）键，调出Dash并输入Terminal。

终端窗口将从你的用户账户的主文件夹开始。如果我的用户名是sweigart，OS X上主文件夹在/Users/asweigart，Linux上在/home/asweigart。波浪纯字符（~）是主文件夹的快捷方式，所以你可以输入cd ~切换到主文件夹。也可以使用cd命令，将当前工作目录改变到任何其他目录。在OS X和Linux上，pwd命令将打印当前工作目录。

为了运行Python程序，将你的.py文件保存到你的主文件夹。然后，更改.py文件的权限，运行chmod +x pythonScript.py，使之成为可执行文件。文件权限超出了本书的范围，但如果你想在终端窗口运行程序，就需要对Python文件运行此命令。这样做之后，当你打开一个终端窗口，输入./pythonScript.py，就能运行该脚本。脚本顶部的#!行会告诉操作系统，在哪里可以找到Python解释器。

B.4 运行Python程序时禁用断言

你可以禁用Python程序中的assert语句，从而稍稍提高性能。从终端窗口运行Python时，在python或python3之后和.py文件之前加上-O开关。这将运行程序的优化版本，跳过断言检查。

附录C　习题答案

本附录包含每章末习题的答案。我强烈建议你花时间解答这些习题。编程不只是记住语法和函数名列表。像学习外语一样，练习越多，收获就越大。有许多网站也包含编程习题。你可以在http://nostarch.com/automatestuff/ 找到这些网站的列表。

第1章

1．操作符是+、−、*和/。值是'hello'、−88.8和5。

2．字符串是'spam'，变量是spam。字符串总是以引号开始和结束。

3．本章介绍的3种数据类型是整数、浮点数和字符串。

4．表达式是值和操作符的结合。所有表达式都求值为（即归约为）一个值。

5．表达式求值为一个值。语句不是这样。

6．bacon变量被设置为20。表达式bacon + 1并没有对bacon重新赋值（重新赋值需要一个赋值语句：bacon =bacon + 1）。

7．两个表达式都求值为字符串'spamspamspam'。

8．变量名不能以数字开始。

9．int()、float() 和str() 函数将返回传入值的整型、浮点型和字符串版本。

10．该表达式导致错误是因为，99是一个整数，只有字符串能用+操作符与其他字符串连接。正确的方式是'I have eaten ' + str(99) + ' burritos.'。

第2章

1．True和False，使用大写的T和F，其他字母是小写。

2．and、or和not。

3．True and True是True。

True and False是False。

False and True是False。

False and False是False。

True or True是True。

True or False是True。

False or True是True。

False or False是False。

not True是False。

not False是True。

4．False

False

True

False

False

True

5．==、!=、<、>、<=和>=。

6．==是等于操作符，它比较两个值，求值为一个布尔值，而=是赋值操作符，将值保存在变量中。

7．条件是一个表达式，它用于控制流语句中，求值为一个布尔值。

8．3个语句块是if语句中的全部内容，以及print（'bacon'）和print（'ham'）这两行。

```
print('eggs')
if spam > 5:
     print('bacon')
else:
     print('ham')
print('spam')
```

9．代码：

```
if spam == 1:
     print('Hello')
elif spam == 2:
     print('Howdy')
else:
     print('Greetings!')
```

10．按Ctrl-C来停止陷在无限循环中的程序。

11．break语句将执行移出循环，接着循环之后执行。continue语句将执行移到循环的开始。

12．它们都是做同样的事。range（10）调用产生的范围是从0直到（但不包括）10，range（0，10）明确告诉循环从0开始，range（0，10，1）明确告诉循环每次迭代让变量增加1。

13．代码：

```
for i in range(1, 11):
```

```
    print(i)
```

以及：

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

14．该函数的调用方式是spam.bacon()。

第3章

1．函数减少了重复的代码。这让程序更短，更容易阅读，更容易修改。

2．函数中的代码在函数被调用时执行，而不是在函数定义时。

3．def语句定义了（即创建了）一个函数。

4．函数包含def语句和在def子句中的代码。函数调用让程序执行转到函数内，函数调用求值为该函数的返回值。

5．在调用一个函数时，创建了一个全局函数和一个局部作用域。

6．函数返回时，局部作用域被销毁，其中所有的变量都被遗忘了。

7．返回值是函数调用求值的结果。像所有值一样，返回值可以作为表达式的一部分。

8．如果函数没有return语句，它的返回值就是None。

9．global语句强制函数中的一个变量引用该全局变量。

10．None的数据类型是NoneType。

11．import语句导入了areallyourpetsnamederic模块（顺便说一句，这不是一个真正的Python模块）。

12．该函数可以通过spam.bacon() 调用。

13．将可能导致错误的代码行放在一个try子句中。

14．可能导致错误的代码放在 try 子句中。发生错误时要执行的代码放在except子句中。

第4章

1．空的列表值，它是一个列表，不包含任何列表项。这类似于''是空的字符串值。

2．spam[2] = 'hello'（注意，列表中的第3个值下标是2，因为第1个值下标是0。）

3．'d'（注意'3' * 2是字符串'33'，它被传入int()，然后再除以11。这最终求值为3。在使用值的地方，都可以使用表达式）。

4．'d'（负数下标从末尾倒数）。

5．['a', 'b']

6．1

7．[3.14, 'cat', 11, 'cat', True, 99]

8．[3.14, 11, 'cat', True]

9．列表连接的操作符是+，复制的操作符是*（这和字符串一样）。

10．append() 只会将值添加在列表末尾，而insert() 可以将值添加在列表的任何位置。

11．del语句和remove() 列表方法是从列表中删除值的两种方法。

12．列表和字符串都可以传入 len()，都有下标和切片，用于 for 循环，连接或复制，并与in和not in操作符一起使用。

13．列表是可以修改的，它们可以添加值、删除值和修改值。元组是不可修改的，它们根本不能改变。而且，元组使用的是括号（和），而列表使用的是方括号 [和]。

14．（42，）（末尾的逗号是必须的）。

15．分别使用tuple() 和list() 函数。

16．它们包含对列表值的引用。

17．copy.copy() 函数将浅拷贝列表，而copy.deepcopy() 函数将深拷贝列表。也就是说，只有copy.deepcopy() 会复制列表内的所有列表。

第5章

1．两个花括号：{}

2．{'foo': 42}

3．保存在字典中的项是无序的，而列表中的项是有序的。

4．会得到KeyError错误。

5．没有区别。in操作符检查一个值是不是字典中的一个键。

6．'cat' in spam检查字典中是不是有一个 'cat' 键，而'cat' in spam.values() 检查是否有一个值 'cat' 对应于spam中的某个键。

7．spam.setdefault('color', 'black')

8．pprint.pprint()

第6章

1．转义字符表示字符串中的一些字符，这些字符用别的方式很难在代码中打出来。

2．\n是换行符，\t是制表符。

3．\\转义字符表示一个反斜杠。

4．Howl's 中的单引号没有问题，因为你用了双引号来标识字符串的开始和结束。

5．多行字符串让你在字符串中使用换行符，而不必用\n转义字符。

6．这些表达式求值为以下值：

- 'e'
- 'Hello'
- 'Hello'
- 'lo world!

7．这些表达式求值为以下值：

- 'HELLO'
- True
- 'hello'

8．这些表达式求值为以下值：

- ['Remember,', 'remember,', 'the', 'fifth', 'of', 'November.']
- 'There-can-be-only-one.'

9．分别用rjust()、ljust() 和center() 字符串方法。

10．lstrip() 和rstrip() 方法分别从字符串的左边和右边移除空白字

符。

第7章

1．re.compile() 函数返回Regex对象。

2．使用原始字符串是为了让反斜杠不必转义。

3．search() 方法返回Match对象。

4．group() 方法返回匹配文本的字符串。

5．分组0是整个匹配，分组1包含第一组括号，分组2包含第二组括号。

6．句号和括号可以用反斜杠转义：.、\（和\）。

7．如果正则表达式没有分组，就返回字符串的列表。如果正则表达式有分组，就返回字符串的元组的列表。

8．| 字符表示匹配两个组中的“任何一个”。

9．? 字符可以表示“匹配前面分组0次或1次”，或用于表示非贪心匹配。

10．+匹配1次或多次。*匹配0次或多次。

11．{3}匹配前面分组的精确3次实例。{3, 5} 匹配3至5次实例。

12．缩写字符分类\d、\w和\s分别匹配一个数字、单词或空白字符。

13．缩写字符分类\D、\W和\S分别匹配一个字符，它不是数字、单词或空白字符。

14．将re.I或re.IGNORECASE作为第二个参数传入re.compile()，让匹配不区分大小写。

15．字符.通常匹配任何字符，换行符除外。如果将re.DOTALL作为第二个参数传入re.compile()，那么点也会匹配换行符。

16．.执行贪心匹配，. ?执行非贪心匹配。

17．[0-9a-z]或[a-z0-9]

18．'X drummers, X pipers, five rings, X hens'

19．re.VERBOSE 参数允许为传入 re.compile() 的字符串添加空格和注释。

20．re.compile（r'^\d{1,3}(,{3})*$'）将创建这个正则表达式，但其他正则表达式字符串可以生成类似的正则表达式。

21．re.compile(r'[A-Z][a-z]*\sNakamoto')

22．re.compile(r'(Alice|Bob|Carol)\s(eats|pets|throws)\s(apples|cats|baseballs).', re.IGNORECASE)

第**8**章

1．相对路径是相对于当前工作目录。

2．绝对路径从根文件夹开始，诸如/或C:\。

3．os.getcwd() 函数返回当前工作目录。os.chdir() 函数改变当前工作目录。

4．文件夹. 是当前文件夹，.. 是父文件夹。

5．C:\bacon\eggs是目录名，而spam.txt是基本名称。

6．字符串 'r' 对应读模式，'w' 对应写模式，'a' 对应添加模式。

7．已有的文件用写模式打开，原有内容会被删除并完全覆写。

8．read() 方法将文件的全部内容作为一个字符串返回。readlines()

返回一个字符串列表，其中每个字符串是文件内容中的一行。

9．shelf值类似字典值，它有键和值，以及keys() 和values() 方法，类似于同名的字典方法。

第9章

1．shutil.copy() 函数将拷贝一个文件，而shutil.copytree() 将拷贝整个文件夹，以及它的所有内容。

2．shutil.move() 函数用于重命名文件，以及文件移动。

3．send2trash函数将一个文件或文件夹移到回收站，而shutil函数将永久地删除文件和文件夹。

4．zipfile.ZipFile() 函数等价于open() 函数，第一个参数是文件名，第二个参数是打开ZIP文件的模式（读、写或添加）。

第10章

1．assert(spam >= 10, 'The spam variable is less than 10.')

2．assert(eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!')或assert(eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!')

3．assert(False, 'This assertion always triggers.')

4．为了能调用logging.debug()，必须在程序开始时加入以下两行：

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

5．为了能利用logging.debug() 将日志消息发送到文件programLog.txt中，必须在程序开始时加入以下两行：

```
import logging
>>> logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

6．DEBUG、INFO、WARNING、ERROR和CRITICAL

7．logging.disable (logging.CRITICAL)

8．可以禁用日志消息，不必删除日志函数调用。可以选择禁用低级别日志消息。可以创建日志消息。日志消息提供了时间戳。

9．Step按扭让调试器进入函数调用。Over按钮将快速执行函数调用，不会单步进入其中。Out按钮将快速执行余下的代码，直到走出当前所处的函数。

10．在点击Go后，调试器将在程序末尾或断点处停止。

11．断点设在一行代码上，在程序执到到达该行时，它导致调试器暂停。

12．要在 IDLE 中设置断点，就在代码行上单击右键，从弹出菜单中选择Set Breakpoint。

第11章

1．webbrowser模块有一个open() 方法，它启动web浏览器，打开指定的URL，就这样。Requests模块可以从网上下载文件和页面。BeautifulSoup模块解析HTML。最后，selenium模块可以启动并控制浏览器。

2．requests.get() 函数返回一个Response对象，它有一个text属性，包含下载内容的字符串。

3．如果下载有问题，raise_for_status() 方法将抛出异常，如果下载成功，什么也不做。

4．Response对象的status_code属性包含了HTTP状态码。

5．以'wb'，即“写二进制”模式在你的计算机上打开新文件后，利用一个 for循环迭代遍历Response对象的iter_content() 方法，将各段写入该文件。下面是例子：

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
     saveFile.write(chunk)
```

6．F12在Chrome中打开开发者工具。按下Ctrl-Shift-C（在Windows和Linux上）或⌘-Option-C（在OS X），在Firefox中打开开发者工具。

7．右键点击页面上的元素，并从菜单中选择Inspect Element。

8．'#main'

9．'.highlight'

10．'div div'

11．'button[value="favorite"]'

12．spam.getText()

13．linkElem.attrs

14．selenium模块是通过from selenium import webdriver导入的。

15．find*element* 方法将第一个匹配的元素返回，作为一个*WebElement*对象。*find* elements 方法返回所有匹配的元素，作为一个WebElement对象列表。

16．click() 和send_keys() 方法分别模拟鼠标点击和键盘按键。

17．对表单中的任意对象调用submit() 方法将提交该表单。

18．forward()、back() 和refresh() 等WebDriver对象方法模拟了这些浏览器按钮。

第12章

1．openpyxl.load_workbook() 函数返回一个Workbook对象。

2．get_sheet_names() 方法返回一个Worksheet对象。

3．调用wb.get_sheet_by_name('Sheet1')。

4．调用wb.get_active_sheet()。

5．sheet['C5'].value或sheet.cell(row=5, column=3).value

6．sheet['C5'] = 'Hello'或sheet.cell(row=5, column=3).value = 'Hello'

7．cell.row和cell.column。

8．它们分别返回表中最高列和最高行的整数值。

9．openpyxl.cell.column_index_from_string('M')

10．openpyxl.cell.get_column_letter(14)

11．sheet['A1':'F1']

12．wb.save('example.xlsx')

13．公式的设置和值一样。将单元格的value属性设置为公式文本的字符串。记住公式以=号开始。

14．在调用load_workbook() 时，传入True作为data_only关键字参数。

15．sheet.row_dimensions[5].height = 100

16．sheet.column_dimensions['C'].hidden = True

17．OpenPyXL 2.0.5不会加载冻结窗格、打印标题、图像或图表。

18．冻结窗格就是总是会出现在屏幕上的行和列。它们作为表头是很有用的。

19．openpyxl.charts.Reference()、openpyxl.charts.Series()、openpyxl.charts. BarChart()、chartObj.append(seriesObj)和add_chart()。

第**13**章

1．File对象由open() 返回。

2．对PdfFileReader() 用读二进制（'rb'），对PdfFileWriter() 用写二进制（'wb'）。

3．调用getPage(4)将返回第5页的Page对象，因为0页就是第1页。

4．在PdfFileReader对象中，numPages变量保存了页数的整数。

5．调用decrypt('swordfish')。

6．rotateClockwise() 和rotateCounterClockwise() 方法。旋转度数作为整数参数传入。

7．docx.Document('demo.docx')

8．文档包含多个段落。段落从一个新行开始，包含多个Run对象。Run对象是段落内连续的字符分组。

9．使用doc.paragraphs。

10．Run对象有这些变量（不是Paragraph）。

11．True总是让Run对象成为粗体，False让它总是不是粗体，不论

样式的粗体设置是什么。None让Run对象使用该样式的粗体设置。

12．调用docx.Document() 函数。

13．doc.add_paragraph('Hello there!')

14．整数0、1、2、3和4。

第14章

1．在Excel中，电子表格的值可以是字符串以外的数据类型，单元格可以有不同的字体、大小或颜色设置，单元格可以有不同的宽度和高度，相邻的单元格可以合并，可以嵌入图像和图表。

2．传入一个File对象，通过调用open() 获得。

3．对于Reader对象，File对象需要以读二进制模式（'rb'）打开，对于Writer对象，需要以写二进制模式（'wb'）打开。

4．writerow() 方法。

5．delimiter参数改变了分隔一行中单元格所用的字符串。lineterminator参数改变了分隔行的字符串。

6．json.loads()

7．json.dumps()

第15章

1．许多日期和时间程序使用的一个参考时刻。该时刻是1970年1月1日，UTC。

2．time.time()

3．time.sleep(5)

4．返回与传入参数最近的整数。例如，round（2.4）返回2。

5．datetime对象表示一个特定的时刻。timedelta对象表示一段时间。

6．threadObj = threading.Thread(target=spam)

7．threadObj.start()

8．确保在一个线程中执行的代码不会和另一个线程中的代码读写相同的变量。

9．subprocess.Popen('c:\Windows\System32\calc.exe')

第16章

1．分别是SMTP和IMAP。

2．smtplib.SMTP()、smtpObj.ehlo()、smptObj.starttls()和smtpObj.login()。

3．imapclient.IMAPClient() and imapObj.login()

4．IMAP关键字的字符串列表，例如'BEFORE <date>'、'FROM <string>'或'SEEN'。

5．将变量imaplib._MAXLINE赋值为一个大整数，例如10000000。

6．pyzmail模块读取下载的邮件。

7．你需要Twilio账户的SID号、认证标识号，以及你的Twilio电话号码。

第17章

1．RGBA值是4个整数的元组，每个整数的范围是0至255。4个整数对应于颜色的红、绿、蓝和alpha值（透明度）。

2．函数调用ImageColor.getcolor（'CornflowerBlue'，'RGBA'）将返

回（100，149，237，255），该颜色的RGBA值。

3．矩形元组是4个整数的元组：分别是左边的x坐标，顶边的y坐标，宽度和高度。

4．Image.open（'zophie.png'）

5．imageObj.size是两个整数的元组，宽度和高度。

6．imageObj.crop((0, 50, 50, 50))。请注意，传入crop() 的是一个矩形元组，不是4个独立的整数参数。

7．调用Image对象的imageObj.save（'new_filename.png'）方法。

8．ImageDraw模块包含在图像上绘画的代码。

9．ImageDraw对象有一些绘制形状的方法，例如point()、line() 或rectangle()。这些对象是将Image对象传入ImageDraw.Draw() 函数后返回的。

第**18**章

1．将鼠标移到屏幕的左上角，即坐标（0，0）。

2．pyautogui.size() 返回2个整数的元组，表示屏幕的宽和高。

3．pyautogui.position() 返回2个整数的元组，表示鼠标的x和y坐标。

4．moveTo() 函数将鼠标移到屏幕的绝对坐标处，而moveRel() 函数相对于鼠标的当前位置来移动鼠标。

5．pyautogui.dragTo() 和pyautogui.dragRel()。

6．pyautogui.typewrite('Hello world!')

7．要么向pyautogui.typewrite() 输入键盘键字符串的列表（例如'left'），要么向pyautogui.press() 输入单个键盘键字符串。

8．pyautogui.screenshot('screenshot.png')

9．pyautogui.PAUSE = 2

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 请输入优惠码 使用优惠码 里填入可使用的积分数值，即可扣减相应金额。

> **特别优惠**
>
> 购买本电子书的读者专享异步社区优惠券 。 使用方法：注册成为社区用户，在下单购书时输入“**57AWG** ”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹（John Z. Sonmez）(作者)　王小刚 (译者)　杨海玲 (责任编辑)

分享　6 推荐　想读　9.0K 阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

纸质版 ¥59.00 ¥46.02（7.8 折）

电子版 ¥35.00

电子版 + 纸质版 ¥59.00

现在购买　下载PDF样章　配套文件下载

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试

身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：368449889

投稿**&**咨询：contact@epubit.com.cn

PACKT PUBLISHING

异步图书 www.epubit.com.cn

使用Python高效爬取页面数据的艺术

精通Python爬虫框架Scrapy

Learning Scrapy

[美] 迪米特里奥斯 考奇斯-劳卡斯（Dimitrios Kouzis-Loukas） 著
李斌 译

 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

目　　录

版权信息

书名：精通Python爬虫框架Scrapy

ISBN：978-7-115-47420-9

本书由人民邮电出版社发行数字版。

- 著　　　　[美]迪米特里奥斯 考奇斯-劳卡斯 (Dimitrios Kouzis-Loukas)

  译　　　　李　斌

  责任编辑　傅道坤

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

Scrapy是使用Python开发的一个快速、高层次的屏幕抓取和Web抓取框架，用于抓Web站点并从页面中提取结构化的数据。本书以Scrapy 1.0版本为基础，讲解了Scrapy的基础知识，以及如何使用Python和三方API提取、整理数据，以满足自己的需求。

本书共11章，其内容涵盖了Scrapy基础知识，理解HTML和XPath，安装Scrapy并爬取一个网站，使用爬虫填充数据库并输出到移动应用中，爬虫的强大功能，将爬虫部署到Scrapinghub云服务器，Scrapy的配置与管理，Scrapy编程，管道秘诀，理解Scrapy性能，使用Scrapyd与实时分析进行分布式爬取。本书附录还提供了各种必备软件的安装与故障排除等内容。

本书适合软件开发人员、数据科学家，以及对自然语言处理和机器学习感兴趣的人阅读。

关于作者

Dimitrios Kouzis-Loukas 作为一位顶级的软件开发人员，已经拥有超过15年的经验。同时，他还使用自己掌握的知识和技能，向广大读者讲授如何编写优秀的软件。

他学习并掌握了多门学科，包括数学、物理学以及微电子学。他对这些学科的透彻理解，提高了自身的标准，而不只是“实用的解决方案”。他知道真正的解决方案应当是像物理学规律一样确定，像ECC内存一样健壮，像数学一样通用。

Dimitrios目前正在使用最新的数据中心技术开发低延迟、高可用的分布式系统。他是语言无关论者，不过对Python、C++和Java略有偏好。他对开源软硬件有着坚定的信念，他希望他的贡献能够造福于各个社区和全人类。

关于审稿人

Lazar Telebak 是一位自由的Web开发人员，专注于使用Python库/框架进行网络爬取和对网页进行索引。

他主要从事于处理自动化和网站爬取以及导出数据到不同格式（包括CSV、JSON、XML和TXT）和数据库（如MongoDB、SQLAlchemy和Postgres）的项目。

他还拥有前端技术和语言的经验，包括HTML、CSS、JS和jQuery。

前言

让我来做一个大胆的猜测。下面的两个故事之一会和你的经历有些相似。

你与Scrapy的第一次相遇是在网上搜索类似“Web scraping Python”的内容时。你快速对其进行了浏览，然后想“这太复杂了吧……我只需要一些简单的东西。”接下来，你使用Requests库开发了一个Python脚本，并且挣扎于Beautiful Soup中，但最终还是完成了很酷的工作。它有些慢，所以你让它整夜运行。你重新启动了几次，忽略了一些不完整的链接和非英文字符，到早上的时候，大部分网站已经“骄傲地”存在你的硬盘中了。然而难过的是，不知什么原因，你不想再看到自己写的代码。当你下一次再想抓取某些东西时，则会直接前往scrapy.org，而这一次文档给了你很好的印象。现在你可以感受到Scrapy能够以优雅且轻松的方式解决了你面临的所有问题，甚至还考虑到了你没有想到的问题。你不会再回头了。

另一种情况是，你与Scrapy的第一次相遇是在进行网络爬取项目的研究时。你需要的是健壮、快速的企业级应用，而大部分花哨的一键式网络爬取工具无法满足需求。你希望它简单，但又有足够的灵活性，能够让你为不同源定制不同的行为，提供不同的输出类型，并且能够以自动化的形式保证24/7可靠运行。提供爬取服务的公司似乎太贵了，你觉得使用开源解决方案比固定供应商更加舒服。从一开始，Scrapy就像一个确定的赢家。

无论你是出于何种目的选择了本书，我都很高兴能够在这本专注于Scrapy的图书中遇到你。Scrapy是全世界爬虫专家的秘密。他们知道如何使用它以节省工作时间，提供出色的性能，并且使他们的主机费用达到最低限度。如果你没有太多经验，但是还想实现同样的结果，那么很不幸的是，Google并没有能够帮到你。网络上大多数Scrapy信息要么太简单低效，要么太复杂。对于那些想要了解如何充分利用Scrapy找到准确、易理解且组织良好的信息的人们来说，本书是非常有必要的。我希望本书能够帮助Scrapy社区进一步发展，并使其得以广泛应用。

本书内容

第**1**章，**Scrapy**简介，介绍本书和Scrapy，可以让你对该框架及本书剩余部分有一个明确的期望。

第**2**章，理解**HTML**和**XPath**，旨在使爬虫初学者能够快速了解Web相关技术以及我们后续将会使用的技巧。

第**3**章，爬虫基础，介绍了如何安装Scrapy，并爬取一个网站。我们通过向你展示每一个行动背后的方法和思路，逐步开发该示例。学习完本章之后，你将能够爬取大部分简单的网站。

第**4**章，从**Scrapy**到移动应用，展示了如何使用我们的爬虫填充数据库并输出给移动应用。本章过后，你将清晰地认识到爬虫在市场方面所带来的好处。

第**5**章，迅速的爬虫技巧，展示了更强大的爬虫功能，包括登录、更快速地抓取、消费API以及爬取URL列表。

第**6**章，部署到**Scrapinghub**，展示了如何将爬虫部署到Scrapinghub的云服务器中，并享受其带来的可用性、易部署以及可控性等特性。

第**7**章，配置与管理，以组织良好的表现形式介绍了大量的Scrapy功能，这些功能可以通过Scrapy配置启用或调整。

第**8**章，**Scrapy**编程，通过展示如何使用底层的Twisted引擎和Scrapy架构对其功能的各个方面进行扩展，将我们的知识带入一个全新的水平。

第**9**章，管道秘诀，提供了许多示例，在这里我们修改了Scrapy的一些功能，在不会造成性能退化的情况下，将数据插入到数据库（比如MySQL、Elasticsearch及Redis）、接口API，以及遗留应用中。

第**10**章，理解**Scrapy**性能，将帮助我们理解Scrapy的时间是如何花费的，以及我们需要怎么做来提升其性能。

第**11**章，使用**Scrapyd**与实时分析进行分布式爬取，这是本书最后一章，展示了如何在多台服务器中使用Scrapyd实现横向扩展，以及如何将爬取得到的数据提供给Apache Spark服务器以执行数据流分析。

阅读本书的前提

为了使本书代码和内容的受众尽可能广泛，我们付出了大量的努力。我们希望提供涉及多服务器和数据库的有趣示例，不过我们并不希望你必须完全了解如何创建它们。我们使用了一个称为Vagrant的伟大

技术，用于在你的计算机中自动下载和创建一次性的多服务器环境。我们的Vagrant配置在Mac OS X和Windows上时使用了虚拟机，而在Linux上则是原生运行。

对于Windows和Mac OS X，你需要一个支持Intel或AMD虚拟化技术（VT-x或AMD-v）的64位计算机。大多数现代计算机都没有问题。对于大部分章节来说，你还需要专门为虚拟机准备1GB内存，不过在第9章和第11章中则需要2GB内存。附录A讲解了安装必要软件的所有细节。

Scrapy本身对硬件和软件的需求更加有限。如果你是一位有经验的读者，并且不想使用Vagrant，也可以根据第 3 章的内容在任何操作系统中安装Scrapy，即使其内存十分有限。

当你成功创建Vagrant环境后，无需网络连接，就可以运行本书几乎全部示例了（第4章和第6章的示例除外）。是的，你可以在航班上阅读本书了。

本书读者

本书尝试着去适应广泛的读者群体。它可能适合如下人群：

- 需要源数据驱动应用的互联网创业者；
- 需要抽取数据进行分析或训练模型的数据科学家与机器学习从业者；
- 需要开发大规模爬虫基础架构的软件工程师；
- 想要为其下一个很酷的项目在树莓派上运行Scrapy的爱好者。

就必备知识而言，阅读本书只需要用到很少的部分。在最开始的几章中，本书为那些几乎没有爬虫经验的读者提供了网络技术和爬虫的基础知识。Python易于阅读，对于有其他编程语言基本经验的任何读者来说，与爬虫相关的章节中给出的大部分代码都很易于理解。

坦率地说，我相信如果一个人在心中有一个项目，并且想使用Scrapy的话，他就能够修改本书中的示例代码，并在几个小时之内良好地运行起来，即使这个人之前没有爬虫、Scrapy或Python经验。

在本书的后半部分中，我们将变得更加依赖于Python，此时初学者可能希望在进一步研究之前，先让自己用几个星期的时间丰富Scrapy的基础经验。此时，更有经验的Python/Scrapy开发者将学习使用Twisted进行事件驱动的Python开发，以及非常有趣的Scrapy内部知识。在性能章节，一些数学知识可能会有用处，不过即使没有，大多数图表也能给我们清晰的感受。

第1章　Scrapy简介

欢迎来到你的Scrapy之旅。通过本书，我们旨在将你从一个只有很少经验甚至没有经验的Scrapy初学者，打造成拥有信心使用这个强大的框架从网络或者其他源爬取大数据集的Scrapy专家。本章将介绍Scrapy，并且告诉你一些可以用它实现的很棒的事情。

1.1　初识Scrapy

Scrapy是一个健壮的网络框架，它可以从各种数据源中抓取数据。作为一个普通的网络用户，你会发现自己经常需要从网站上获取数据，使用类似Excel的电子表格程序进行浏览（参见第3章），以便离线访问数据或者执行计算。而作为一个开发者，你需要经常整合多个数据源的数据，但又十分清楚获得和抽取数据的复杂性。无论难易，Scrapy都可以帮助你完成数据抽取的行动。

以健壮而又有效的方式抽取大量数据，Scrapy已经拥有了多年经验。使用Scrapy，你只需一个简单的设置，就能完成其他爬虫框架中需要很多类、插件和配置项才能完成的工作。快速浏览第7章，你就能体会到通过简单的几行配置，Scrapy可以实现多少功能。

从开发者的角度来说，你也会十分欣赏Scrapy的基于事件的架构（我们将在第8章和第9章中对其进行深入探讨）。它允许我们将数据清洗、格式化、装饰以及将这些数据存储到数据库中等操作级联起来，只

要我们操作得当，性能降低就会很小。在本书中，你将学会怎样可以达到这一目的。从技术上讲，由于Scrapy是基于事件的，这就能够让我们在拥有上千个打开的连接时，可以通过平稳的操作拆分吞吐量的延迟。来看这样一个极端的例子，假设你需要从一个拥有汇总页的网站中抽取房源，其中每个汇总页包含100个房源。Scrapy可以非常轻松地在该网站中并行执行16个请求，假设完成一个请求平均需要花费1秒钟的时间，你可以每秒爬取16个页面。如果将其与每页的房源数相乘，可以得出每秒将产生1600个房源。想象一下，如果每个房源都必须在大规模并行云存储当中执行一次写入，每次写入平均需要耗费3秒钟的时间（非常差的主意）。为了支持每秒16个请求的吞吐量，就需要我们并行运行1600 × 3 = 4800次写入请求（你将在第9章中看到很多这样有趣的计算）。对于一个传统的多线程应用而言，则需要转变为4800个线程，无论是对你，还是对操作系统来说，这都会是一个非常糟糕的体验。而在Scrapy的世界中，只要操作系统没有问题，4800个并发请求就能够处理。此外，Scrapy的内存需求和你需要的房源数据量很接近，而对于多线程应用而言，则需要为每个线程增加与房源大小相比十分明显的开销。

简而言之，缓慢或不可预测的网站、数据库或远程API都不会对Scrapy的性能产生毁灭性的结果，因为你可以并行运行多个请求，并通过单一线程来管理它们。这意味着更低的主机托管费用，与其他应用的协作机会，以及相比于传统多线程应用而言更简单的代码（无同步需求）。

1.2　喜欢Scrapy的更多理由

Scrapy已经拥有超过5年的历史了，成熟而又稳定。除了上一节中提到的性能优势外，还有下面这些能够让你爱上Scrapy的理由。

- Scrapy能够识别残缺的HTML

你可以在Scrapy中直接使用Beautiful Soup或lxml，不过Scrapy还提供了一种在lxml之上更高级的XPath（主要）接口——**selectors** 。它能够更高效地处理残缺的HTML代码和混乱的编码。

- 社区

Scrapy拥有一个充满活力的社区。只需要看看https://groups.google.com/ forum/#!forum/scrapy-users 上的邮件列表，以及Stack Overflow网站（http:// stackoverflow.com/questions/tagged/ scrapy）中的上千个问题就可以知道了。大部分问题都能够在几分钟内得到回应。更多社区资源可以从http://scrapy.org/ community/中获取到。

- 社区维护的组织良好的代码

Scrapy要求以一种标准方式组织你的代码。你只需编写被称为爬虫和管道的少量Python模块，并且还会自动从引擎自身获取到未来的任何改进。如果你在网上搜索，可以发现有相当多专业人士拥有Scrapy经验。也就是说，你可以很容易地找到人来维护或扩展你的代码。无论是谁加入你的团队，都不需要漫长的学习曲线，来理解你的自定义爬虫中的特别之处。

- 越来越多的高质量功能

如果你快速浏览发布日志（http://doc.scrapy.org/en/latest/news.html），就会注意到无论是在功能上，还是在稳定性/bug修复上，Scrapy都在不断地成长。

1.3　关于本书：目标和用途

在本书中，我们的目标是通过重点示例和真实数据集教你使用Scrapy。大部分章节将专注于爬取一个示例的房屋租赁网站。我们选择这个例子，是因为它能够代表大多数的网站爬取项目，既能让我们介绍感兴趣的变动，又不失简单。以该示例为主题，可以帮助我们聚焦于Scrapy，而不会分心。

我们将从只运行几百个页面的小爬虫开始，最终在第11章中使用几分钟的时间，将其扩展为能够处理5万个页面的分布式爬虫。在这个过程中，我们将向你介绍如何将Scrapy与MySQL、Redis和Elasticsearch等服务相连接，使用Google的地理编码API找到我们示例属性中的位置坐标，以及向Apache Spark提供数据用于预测最影响房价的关键词。

你需要做好阅读本书多次的准备。你可能需要从略读开始，先理解其架构。然后阅读一到两章，仔细学习、实验一段时间，再进入后面的章节。如果你觉得自己已经熟悉了某一章的内容，那么跳过这一章也无需担心。尤其是如果你已经了解HTML和XPath，那么就没有必要花费太多时间在第2章上面了。不用担心，对你来说本书还有很多需要学习的内容。一些章节，比如第8章，将参考书和教程的元素结合起来，深入编程概念。这就是一个例子，我们可能会阅读某一章几次，在这中间允许我们有几个星期的时间实践Scrapy。你在继续阅读后续的章节，比

如以应用为主的第9章之前，不需要完美掌握第8章中的内容。阅读后续的内容，有助于你理解如何使用编程概念，如果你愿意的话，可以回过头来反复阅读几次。

为使本书既有趣，又对初学者友好，我们已经试图做了平衡。不过我们不会做的一件事情是，在本书中教授Python。对于这一主题，目前已经有了很多优秀的书籍，不过我更加建议的是以一种轻松的心态来学习。Python如此流行的一个理由是因为它比较简单、整洁，并且阅读起来更近似于英文。Scrapy是一个高级框架，无论是初学者还是专家，都需要学习。你可以将其称之为“Scrapy语言”。因此，我会推荐你通过材料来学习Python，如果你发觉自己对于Python的语法比较迷惑，那么可以通过一些Python的在线教程或Coursera等为Python初学者开设的免费在线课程予以补充。请放心，即使你不是Python专家，也能够成为一名优秀的Scrapy开发者。

1.4　掌握自动化数据爬取的重要性

对于大多数人来说，掌握一门像Scrapy这样很酷的技术所带来的好奇心和精神上的满足，足以激励我们。令人惊喜的是，在学习这个优秀框架的同时，我们还能享受到开发过程始于数据和社区，而不是代码所带来的好处。

1.4.1　开发健壮且高质量的应用，并提供合理规划

为了开发现代化的高质量应用，我们需要真实的大数据集，如果可能的话，在开始动手写代码之前就应该进行这一步。现代化软件开发就

是实时处理大量不完善数据，并从中提取出知识和有价值的情报。当我们开发软件并应用于大数据集时，一些小的错误和疏忽难以被检测出来，就有可能导致昂贵的错误决策。比如，在做人口统计学研究时，很容易发生仅仅是由于州名过长导致数据被默认丢弃，造成整个州的数据被忽视的错误。在开发阶段，甚至更早的设计探索阶段，通过细心抓取，并使用具有生产质量的真实世界大数据集，可以帮助我们发现和修复错误，做出明智的工程决策。

另外一个例子是，假设你想要设计Amazon风格的“如果你喜欢这个商品，也可能喜欢那个商品”的推荐系统。如果你能够在开始之前，先爬取并收集真实世界的数据集，就会很快意识到有关无效条目、停产商品、重复、无效字符以及偏态分布引起的性能瓶颈等问题。这些数据将会强迫你设计足够健壮的算法，无论是数千人购买过的商品，还是零销售量的新条目，都能够很好地处理。而孤立的软件开发，可能会在几个星期的开发之后，也要面对这些丑陋的真实世界数据。虽然这两种方法最终可能会收敛，但是为你提供进度预估承诺的能力以及软件的质量，都将随着项目进展而产生显著差别。从数据开始，能够带给我们更加愉悦并且可预测的软件开发体验。

1.4.2　快速开发高质量最小可行产品

对于初创公司而言，大规模真实数据的集甚至更加必要。你可能听说过“精益创业”，这是由*Eric Ries* 创造的一个术语，用于描述类似技术初创公司这样极端不确定条件下的业务发展过程。该框架的一个关键概念是最小可行产品（**Minimum Viable Product**，**MVP**），这种产品只有有限的功能，可以被快速开发并向有限的客户发布，用于测试反响及

验证业务假设。基于获得的反馈，初创公司可能会选择继续更进一步的投资，也可能是转向其他更有前景的方向。

在该过程中的某些方面，很容易忽视与数据紧密连接的问题，这正是Scrapy所能为我们做的部分。比如，当邀请潜在的客户尝试使用我们的手机应用时，作为开发者或企业主，会要求他们评判这些功能，想象应用在完成时看起来应该如何。对于这些并非专家的人而言，这里需要的想象有可能太多了。这个差距相当于一个应用只展示了“产品1”、“产品2”、“用户433”，而另一个应用提供了“三星 UN55J6200 55英寸电视机”、用户“Richard S”给出了五星好评以及能够让你直达产品详情页面（尽管事实上我们还没有写这个页面）的有效链接等诸多信息。人们很难客观判断一个MVP产品的功能性，除非使用了真实且令人兴奋的数据。

一些初创企业将数据作为事后考虑的原因之一是认为收集这些数据需要昂贵的代价。的确，我们通常需要开发表单及管理界面，并花费时间录入数据，但我们也可以在编写代码之前使用Scrapy爬取一些网站。在第4章中，你可以看到一旦拥有了数据，开发一个简单的手机应用会有多么容易。

1.4.3　Google不会使用表单，爬取才能扩大规模

当谈及表单时，让我们来看下它是如何影响产品增长的。想象一下，如果Google的创始人在创建其引擎的第一个版本时，包含了一个每名网站管理员都需要填写的表单，要求他们把网站中每一页的文字都复制粘贴过来。然后，他们需要接受许可协议，允许Google处理、存储和展示他们的内容，并剔除大部分广告利润。你能想象解释该想法并说服

人们参与这一过程所需花费的时间和精力会有多大吗？即使市场非常渴望一个优秀的搜索引擎（事实正是如此），这个引擎也不会是Google，因为它的增长过于缓慢。即使是最复杂的算法，也不能弥补数据的缺失。Google使用网络爬虫技术，在页面间跳转链接，填充其庞大的数据库。网站管理员则不需要做任何事情。实际上，反而还需要一些努力才能阻止Google索引你的页面。

虽然Google使用表单的想法听起来有些荒谬，但是一个典型的网站需要用户填写多少表单呢？登录表单、新房源表单、结账表单，等等。这些表单中有多少会阻碍应用增长呢？如果你充分了解你的受众/客户，很可能已经拥有关于他们通常使用并且很可能已经有账号的其他网站的线索了。比如，一个开发者很可能拥有Stack Overflow和GitHub的账号。那么，在获得他们允许的情况下，你是否能够抓取这些站点，只需他们提供给你用户名，就能自动填充照片、简介和一小部分近期文章呢？你能否对他们最感兴趣的一些文章进行快速文本分析，并根据其调整网站的导航结构，以及建议的产品和服务呢？我希望你能够看到如何使用自动化数据抓取替代表单，从而更好地服务你的受众，增长网站规模。

1.4.4　发现并融入你的生态系统

抓取数据自然会让你发现并考虑与你付出相关的社区的关系。当你抓取一个数据源时，很自然地就会产生一些问题：我是否相信他们的数据？我是否相信获取数据的公司？我是否需要和他们沟通以获得更正式的合作？我和他们是竞争关系还是合作关系？从其他源获得这些数据会花费我多少钱？无论如何，这些商业风险都是存在的，不过抓取过程可

以帮助我们尽早意识到这些风险，并制定出缓解策略。

你还会发现自己想知道能够为这些网站和社区带来的回馈是什么。如果你能够给他们带来免费的流量，他们应该会很高兴。另一方面，如果你的应用不能给你的数据源带来一些价值，那么你们的关系可能会很短暂，除非你与他们沟通，并找到合作的方式。通过从不同源获取数据，你需要准备好开发对现有生态系统更友好的产品，充分尊重已有的市场参与者，只有在值得努力时才可以去破坏当前的市场秩序。现有的参与者也可能会帮助你成长得更快，比如你有一个应用，使用两到三个不同生态系统的数据，每个生态系统有10万个用户，你的服务可能最终将这30万个用户以一种创造性的方式连接起来，从而使每个生态系统都获益。例如，你成立了一个初创公司，将摇滚乐与T恤印花社区关联起来，你的公司最终将成为两种生态系统的融合，你和相应的社区都将从中获益并得以成长。

1.5　在充满爬虫的世界里做一个好公民

当开发爬虫时，还有一些事情需要清楚。不负责任的网络爬虫会令人不悦，甚至在某些情况下是违法的。有两个非常重要的事情是避免类似拒绝服务（**DoS**）攻击的行为以及侵犯版权。

对于第一种情况，一个典型的访问者可能每几秒访问一个新的页面。而一个典型的网络爬虫则可能每秒下载数十个页面。这样就比典型用户产生的流量多出了10倍以上。这可能会使网站所有者非常不高兴。请使用流量限速将你产生的流量减少到可以接受的普通用户的水平。此外，还应该监控响应时间，如果发现响应时间增加了，就需要降低爬虫

的强度。好消息是Scrapy对于这些功能都提供了开箱即用的实现（参见第7章）。

对于版权问题，显然你需要看一下你抓取的每个网站的版权声明，并确保你理解其允许做什么，不允许做什么。大多数网站都允许你处理其站点的信息，只要不以自己的名义重新发布即可。在你的请求中，有一个很好的`User-Agent`字段，它可以让网站管理员知道你是谁，你用他们的数据做什么。Scrapy在制造请求时，默认使用`BOT_NAME`参数作为`User-Agent`。如果`User-Agent`是一个URL或者能够指明你的应用名称，那么网站管理员可以通过访问你的站点，更多地了解你是如何使用他们的数据的。另一个非常重要的方面是，请允许任何网站管理员阻止你访问其网站的指定区域。对于基于Web标准的`robots.txt`文件（参见http://www.google.com/robots.txt 的文件示例），Scrapy提供了用于尊重网站管理员设置的功能（`RobotsTxtMiddleware`）。最后，最好向网站管理员提供一些方法，让他们能说明不希望在你的爬虫中出现的东西。至少网站管理员必须能够很容易地找到和你交流及表达顾虑的方式。

1.6　Scrapy不是什么

最后，很容易误解Scrapy可以为你做什么，主要是因为数据抓取这个术语与其相关术语有些模糊，很多术语是交替使用的。我将尝试使这些方面更加清楚，以防止混淆，为你节省一些时间。

Scrapy不是Apache Nutch，也就是说，它不是一个通用的网络爬虫。如果Scrapy访问一个一无所知的网站，它将无法做出任何有意义的

事情。Scrapy是用于提取结构化信息的，需要人工介入，设置合适的XPath或CSS表达式。而Apache Nutch则是获取通用页面并从中提取信息，比如关键字。它可能更适合于一些应用，但对另一些应用则又更不适合。

Scrapy不是Apache Solr、Elasticsearch或Lucene，换句话说，就是它与搜索引擎无关。Scrapy并不打算为你提供包含“Einstein”或其他单词的文档的参考。你可以使用Scrapy抽取数据，然后将其插入到Solr或Elasticsearch当中，我们会在第9章的开始部分讲解这一做法，不过这仅仅是使用Scrapy的一个方法，而不是嵌入在Scrapy内的功能。

最后，Scrapy不是类似MySQL、MongoDB或Redis的数据库。它既不存储数据，也不索引数据。它只用于抽取数据。即便如此，你可能会将Scrapy抽取得到的数据插入到数据库当中，而且它对很多数据库也都有所支持，能够让你的生活更加轻松。然而Scrapy终究不是一个数据库，其输出也可以很容易地更改为只是磁盘中的文件，甚至什么都不输出——虽然我不确定这有什么用。

1.7　本章小结

本章介绍了Scrapy，给出了它能够帮你做什么的概述，并描述了我们认为的使用本书的正确方式。本章还提供了几种自动化数据抓取的方式，通过帮你快速开发能够与现有生态系统更好融合的高质量应用而获益。下一章将介绍HTML和XPath，这是两个非常重要的Web语言，我们在每个Scrapy项目中都将用到它们。

第2章　理解HTML和XPath

为了从网页中抽取信息，你必须对其结构有更多了解。我们将快速浏览HTML、HTML的树状表示，以及在网页上选取信息的一种方式XPath。

2.1　HTML、DOM树表示以及XPath

让我们花费一些时间来了解从用户在浏览器中输入URL（或者更常见的是，在其单击链接或书签时）到屏幕上显示出页面的过程。从本书的视角来看，该过程包含4个步骤，如图2.1所示。

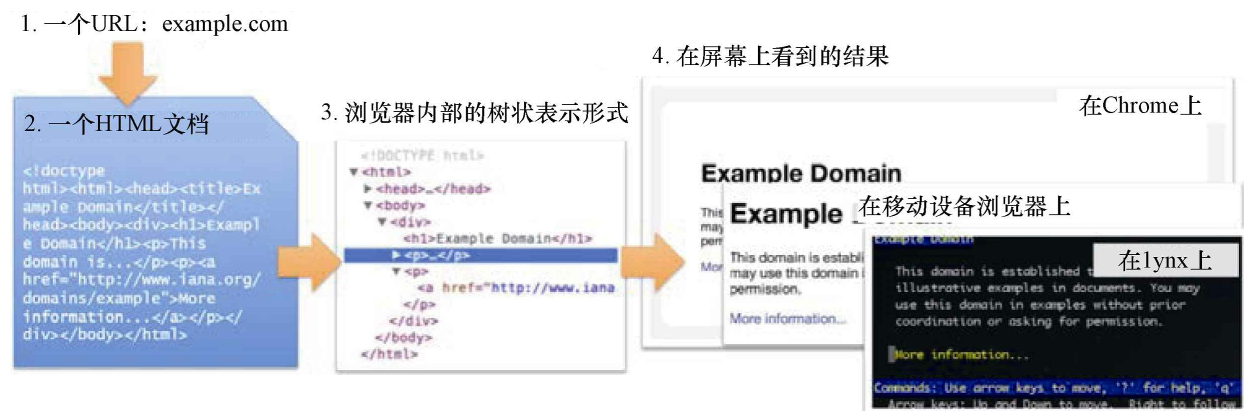


图2.1

- 在浏览器中输入URL。URL的第一部分（域名，比如`gumtree.com`）用于在网络上找到合适的服务器，而URL以及cookie等其他数据则构成了一个请求，用于发送到那台服务器当中。
- 服务端回应，向浏览器发送一个HTML页面。需要注意的是，服务

端也可能返回其他格式，比如XML或JSON，不过目前我们只关注HTML。

- 将HTML转换为浏览器内部的树状表示形式：文档对象模型（**Document Object Model**，**DOM**）。
- 基于一些布局规则渲染内部表示，达到你在屏幕上看到的视觉效果。

下面来看看这些步骤，以及它们所需的文档表示。这将有助于定位你想要抓取并编写程序获取的文本。

2.1.1　URL

对于我们而言，URL分为两个主要部分。第一个部分通过域名系统（**Domain Name System**，**DNS**）帮助我们在网络上定位合适的服务器。比如，当在浏览器中发送`https://mail.google.com/mail/u/0/#inbox`时，将会创建一个对`mail.google.com`的DNS请求，用于确定合适的服务器IP地址，如`173.194.71.83`。从本质上来看，`https://mail.google.com/mail/u/0/#inbox`被翻译为`https://173.194.71.83/mail/ u/0/#inbox`。

URL的剩余部分对于服务端理解请求是什么非常重要。它可能是一张图片、一个文档，或是需要触发某个动作的东西，比如向服务器发送邮件。

2.1.2　HTML文档

服务端读取URL，理解我们的请求是什么，然后回应一个HTML文档。该文档实质上就是一个文本文件，我们可以使用TextMate、Notepad、vi或Emacs打开它。和大多数文本文档不同，HTML文档具有由万维网联盟指定的格式。该规范当然已经超出了本书的范畴，不过还是让我们看一个简单的HTML页面。当访问`http://example.com`时，可以在浏览器中选择**View Page Source**（查看页面源代码）以看到与其相关的HTML文件。在不同的浏览器中，具体的过程是不同的；在许多系统中，可以通过右键单击找到该选项，并且大部分浏览器在你按下*Ctrl* + *U*快捷键（或Mac系统中的*Cmd* + *U*）时可以显示源代码。



在一些页面中，该功能可能无法使用。此时，需要通过单击Chrome菜单，然后选择**Tools** | **View Source**才可以。

下面是`http://example.com`目前的HTML源代码。

```
<!doctype html>
<html>
  <head>
      <title>Example Domain</title>
      <meta charset="utf-8" />
      <meta http-equiv="Content-type"
              content="text/html; charset=utf-8" />
      <meta name="viewport" content="width=device-width,
              initial-scale=1" />
      <style type="text/css"> body { background-color: ...
              } }</style>
  <body>
      <div>
              <h1>Example Domain</h1>
              <p>This domain is established to be used for
```

```
                  illustrative examples examples in documents.
                  You may use this domain in examples without
                  prior coordination or asking for permission.</p>
               <p><a href="http://www.iana.org/domains/example

">
                  More information...</a></p>
      </div>
  </body>
</html>
```

我将这个HTML文档进行了格式化，使其更具可读性，而你看到的情况可能是所有文本在同一行中。在HTML中，空格和换行在大多数情况下是无关紧要的。

尖括号中间的文本（比如<html> 或<head> ）被称为标签。<html> 是起始标签，而</html> 是结束标签。这两种标签的唯一区别是/字符。这说明，标签是成对出现的。虽然一些网页对于结束标签的使用比较粗心（比如，为独立的段落使用单一的<p> 标签），但是浏览器有很好的容忍度，并且会尝试推测结束的</p> 标签应该在哪里。

<p> 和</p> 标签中的所有东西被称为HTML 元素。请注意，元素中可能还包括其他元素，比如示例中的<div> 元素，或是包含<a> 元素的第二个<p> 元素。

有些标签会更加复杂，比如<a

href="http://www.iana.org/domains/example ">。含有URL的href部分被称为属性。

最后，许多元素还包含文本，比如<h1>元素中的"Example Domain"。

对于我们来说，好消息是这些标签并不都是重要的。唯一可见的东西是body元素中的元素，即<body>和</body>标签之间的元素。<head>部分对于指明诸如字符编码的元信息来说非常重要，不过Scrapy能够处理大部分此类问题，所以很多情况下不需要关注HTML页面的这个部分。

2.1.3　树表示法

每个浏览器都有其自身复杂的内部数据结构，凭借它来渲染网页。DOM表示法具有跨平台、语言无关性等特点，并且被大多数浏览器所支持。

想要在Chrome中查看网页的树表示法，可以右键单击你感兴趣的元素，然后选择**Inspect Element**。如果该功能被禁用，你仍然可以通过单击Chrome菜单并选择**Tools | Developer Tools**来访问它，如图2.2所示。



图2.2

此时，你可以看到一些看起来和HTML表示非常相似但又不完全相同的东西。它就是HTML代码的树表示法。如果不管原始HTML文档是如何使用空格和换行符的话，它看起来几乎就是一样的。你可以单击每个元素，检查或调整属性等，同时可以在屏幕上观察这些变动有何影响。比如，当你双击某个文本，修改它，并按下回车键时，屏幕上的文本将会更新为这个新值。在右侧的**Properties** 标签下，可以看到这个树表示法的属性，并且在底部可以看到一个类似面包屑的结构，它显示出了当前选择的元素在HTML元素层次结构中的确切位置，如图2.3所示。

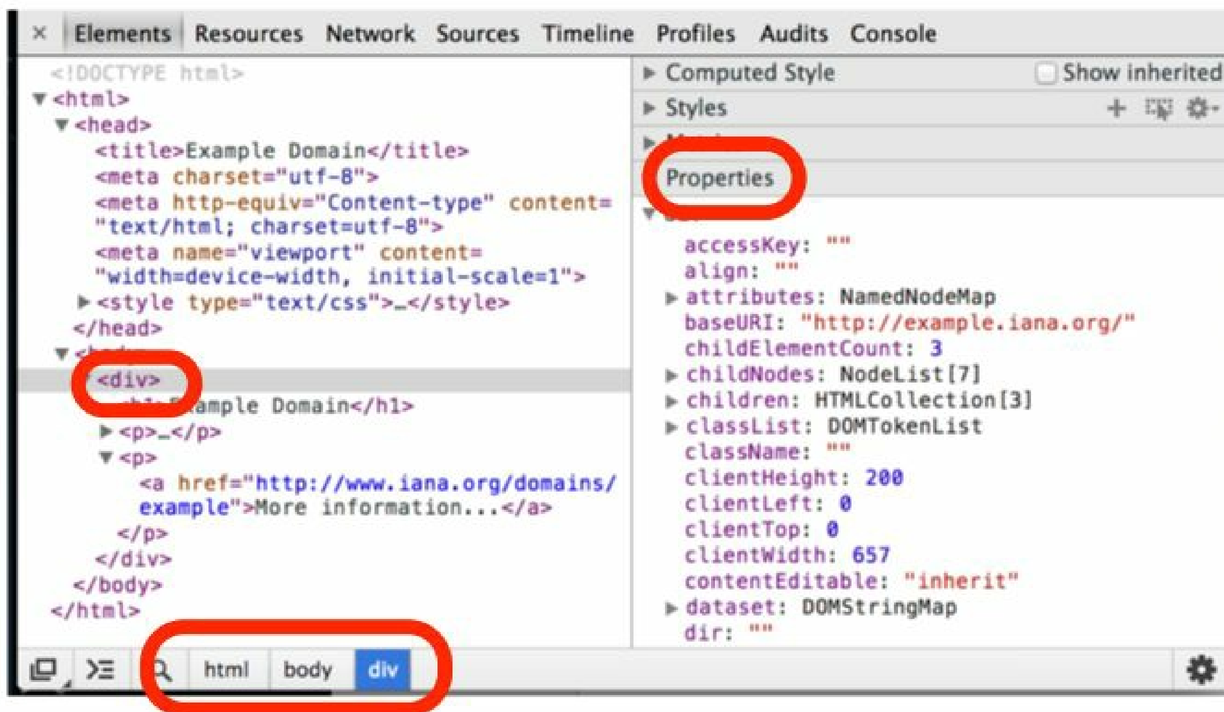


图2.3

需要注意的一个重要事情是，HTML只是文本，而树表示法是浏览器内存里的对象，你可以通过编程的方式查看并操纵它，比如在Chrome中使用**Developer Tools** 。

2.1.4 你会在屏幕上看到什么

HTML文本表示和树表示并不包含任何像我们通常在屏幕上看到的那种漂亮视图。这实际上是HTML成功的原因之一。它应该是一个由人类阅读的文档，并且可以指定页面中的内容，而不是用于在屏幕中渲染的方式。这意味着选择HTML文档并使其更加好看是浏览器的责任，不管它是诸如Chrome的全功能浏览器、移动设备浏览器，甚至是诸如Lynx的纯文本浏览器。

也就是说，网络的发展促使Web开发者和用户对网页渲染的控制产

生了巨大需求。CSS的创建就是为了对HTML元素如何渲染给予提示。不过，对于抓取而言，我们并不需要任何和CSS相关的东西。

那么，树表示法是如何映射到我们在屏幕上所看到的东西呢？答案就是框模型。正如DOM树元素可以包含其他元素或文本一样，默认情况下，当在屏幕上渲染时，每个元素的框表示同样也都包含其嵌入元素的框表示。从这种意义上说，我们在屏幕上所看到的是原始HTML文档的二维表示——树结构也以一种隐藏的方式作为该表示的一部分。比如，在图2.4中，我们可以看到3个DOM元素（一个`<div>`和两个嵌入元素`<h1>`和`<p>`）是如何在浏览器和DOM中呈现的。

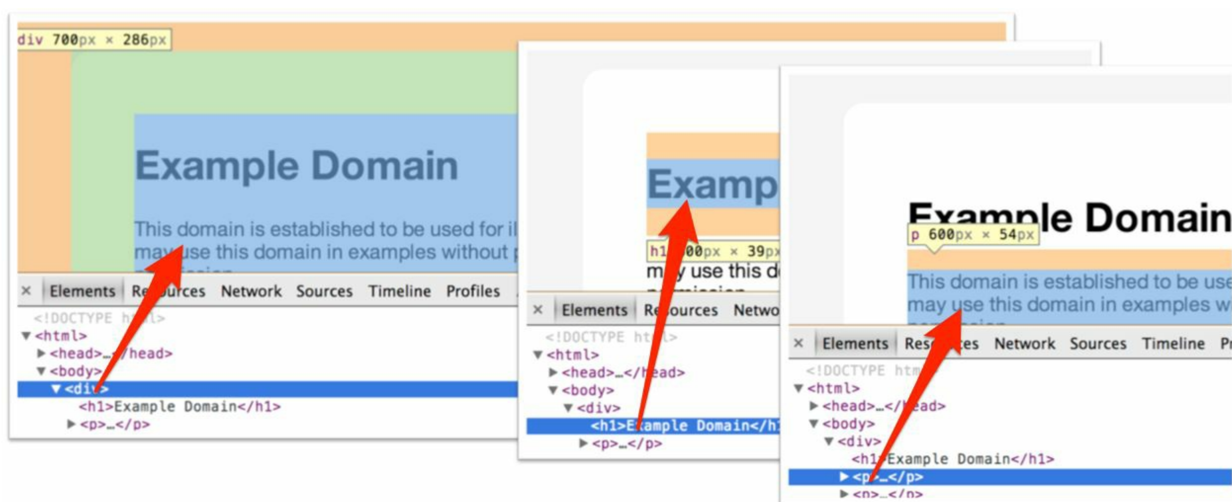


图2.4

2.2　使用XPath选择HTML元素

如果你具有传统软件工程背景，并且不了解XPath相关知识的话，可能会担心为了访问HTML文档中的信息，你将需要做很多字符串匹配、在文档中搜索标签、处理特殊情况等工作，或是需要设法解析整个

树表示法以获取你想抽取的东西。有一个好消息是这些工作都不是必需的。你可以通过一种称为XPath的语言选择并抽取元素、属性和文本，这种语言正是专门为此而设计的。

为了在Google Chrome浏览器中使用XPath，需要单击**Developer Tools** 的**Console** 标签，并使用`$x` 工具函数。比如，你可以尝试在`http://example. com/` 上使用`$x('//h1')` 。它将会把浏览器移动到`<h1>` 元素上，如图2.5所示。



图2.5

你在Chrome的**Console** 标签中将会看到返回的是一个包含选定元素的JavaScript数组。如果将鼠标指针移动到这些属性上，被选取的元素将会在屏幕上高亮显示，这样就会十分方便。

2.2.1　有用的XPath表达式

文档的层次结构始于`<html>` 元素，可以使用元素名和斜线来选择文档中的元素。比如，下面是几种表达式从`http://example.com` 页面

返回的结果。

```
$x('/html')
  [ <html>...</html> ]
$x('/html/body')
  [ <body>...</body> ]
$x('/html/body/div')
  [ <div>...</div> ]
$x('/html/body/div/h1')
  [ <h1>Example Domain</h1> ]
$x('/html/body/div/p')
  [ <p>...</p>, <p>...</p> ]
$x('/html/body/div/p[1]')
  [ <p>...</p> ]
$x('/html/body/div/p[2]')
  [ <p>...</p> ]
```

需要注意的是，因为在这个特定页面中，`<div>` 下包含两个`<p>` 元素，因此`html/body/div/p` 会返回两个元素。可以使用`p[1]` 和`p[2]` 分别访问第一个和第二个元素。

另外还需要注意的是，从抓取的角度来说，文档标题可能是`head` 部分中我们唯一感兴趣的元素，该元素可以通过下面的表达式进行访问。

```
$x('//html/head/title')
  [ <title>Example Domain</title> ]
```

对于大型文档，可能需要编写一个非常大的XPath表达式以访问指定元素。为了避免这一问题，可以使用`//` 语法，它可以让你取得某一

特定类型的元素，而无需考虑其所在的层次结构。比如，//p 将会选择所有的p 元素，而//a 则会选择所有的链接。

```
$x('//p')
  [ <p>...</p>, <p>...</p> ]
$x('//a')
  [ <a href="http://www.iana.org/domains/example




">More
information...</a> ]
```

同样，//a 语法也可以在层次结构中的任何地方使用。比如，要想找到div 元素下的所有链接，可以使用//div//a 。需要注意的是，只使用单斜线的//div/a 将会得到一个空数组，这是因为在example.com中，'div'元素的直接下级中并没有任何'a'元素：

```
$x('//div//a')
  [ <a href="http://www.iana.org/domains/example




">More
information...</a> ]
$x('//div/a')
  [ ]
```

还可以选择属性。http://example.com/中的唯一属性是链接中的href，可以使用符号@来访问该属性，如下面的代码所示。

```
$x('//a/@href')
  [ href="http://www.iana.org/domains/example




" ]
```



实际上，在Chrome的最新版本中，`@href` 不再返回URL，而是返回一个空字符串。不过不用担心，你的XPath表达式仍然是正确的。

还可以通过使用`text()`函数，只选取文本。

```
$x('//a/text()')
  [ "More information..." ]
```

可以使用*符号来选择指定层级的所有元素。比如：

```
$x('//div/*')
[ <h1>Example Domain</h1>, <p>...</p>, <p>...</p> ]
```

你将会发现选择包含指定属性（比如@class ）或是属性为特定值的元素非常有用。可以使用更高级的谓词来选取元素，而不再是前面例子中使用过的p[1] 和p[2] 。比如，//a[@href] 可以用来选择包含href 属性的链接，

而//a[@href="http://www.iana.org/domains/example "] 则是选择href 属性为特定值的链接。

更加有用的是，它还拥有找到href 属性中以一个特定子字符串起始或包含的能力。下面是几个例子。

```
$x('//a[@href]')
  [ <a href="http://www.iana.org/domains/example




">More information...</a> ]
$x('//a[@href="http://www.iana.org/domains/example




"]')
  [ <a href="http://www.iana.org/domains/example




">More information...</a> ]
$x('//a[contains(@href, "iana")]')
  [ <a href="http://www.iana.org/domains/example




">More information...</a> ]
```

```
$x('//a[starts-with(@href, "http://www.")]')
  [ <a href="http://www.iana.org/domains/example




">More information...</a>]
$x('//a[not(contains(@href, "abc"))]')
  [ <a href="http://www.iana.org/domains/example




">More information...</a>]
```

XPath有很多像not()、contains()和starts-with()这样的函数，你可以在线文档（http://www.w3schools.com/xsl/xsl_functions.asp）中找到它们，不过即使不使用这些函数，你也可以走得很远。

现在，我还要再多说一点，大家可以在Scrapy命令行中使用同样的XPath表达式。要打开一个页面并访问Scrapy命令行，只需要输入如下命令：

```
scrapy shell http://example.com
```

在命令行中，可以访问很多在编写爬虫代码时经常需要用到的变量（参见下一章）。这其中最重要的就是响应，对于HTML文档来说就是`HtmlResponse`类，该类可以让你通过`xpath()`方法模拟Chrome中的`$x`。下面是一些示例。

```
response.xpath('/html').extract()
  [u'<html><head><title>...</body></html>']
response.xpath('/html/body/div/h1').extract()
  [u'<h1>Example Domain</h1>']
response.xpath('/html/body/div/p').extract()
  [u'<p>This domain ... permission.</p>', u'<p><a href="http://www.
iana.org/domains/example





">More information...</a></p>']
response.xpath('//html/head/title').extract()
  [u'<title>Example Domain</title>']
response.xpath('//a').extract()
  [u'<a href="http://www.iana.org/domains/example





">More
information...</a>']
response.xpath('//a/@href').extract()
  [u'http://www.iana.org/domains/example





']
response.xpath('//a/text()').extract()
  [u'More information...']
response.xpath('//a[starts-with(@href, "http://www.")]').extract()
  [u'<a href="http://www.iana.org/domains/example





">More
```

```
information...</a>']
```

这就意味着，你可以使用Chrome开发XPath表达式，然后在Scrapy爬虫中使用它们，正如我们在下一节中将要看到的那样。

2.2.2　使用Chrome获取XPath表达式

Chrome通过向我们提供一些基本的XPath表达式，从而对开发者更加友好。从前文提到的检查元素开始：右键单击想要选取的元素，然后选择**Inspect Element**。该操作将会打开**Developer Tools**，并且在树表示法中高亮显示这个HTML元素。现在右键单击这里，在菜单中选择**Copy XPath**，此时XPath表达式将会被复制到剪贴板中。上述过程如图2.6所示。



图2.6

你可以和之前一样，在命令行中测试该表达式。

```
$x('/html/body/div/p[2]/a')
  [ <a href="http://www.iana.org/domains/example

">More
information...</a>]
```

2.2.3 常见任务示例

有一些XPath表达式，你将会经常遇到。让我们看一些目前在维基百科页面上的例子。维基百科拥有一套非常稳定的格式，所以我认为它们不会很快发生改变，不过改变终究还是会发生的。我们把如下这些表达式作为说明性示例。

- 获取id 为"firstHeading" 的h1 标签下span 中的text 。

```
//h1[@id="firstHeading"]/span/text()
```

- 获取id 为"toc "的div 标签内的无序列表（ul ）中所有链接URL
。

```
//div[@id="toc"]/ul//a/@href
```

- 获取class 属性包含"ltr" 以及class 属性包含"skin-vector" 的任意元素内所有标题元素（h1 ）中的文本。这两个字符串可能在同一个class 中，也可能在不同的class 中。

```
//*[contains(@class,"ltr") and contains(@class,"skin-vector")]//h1//text()
```

实际上，你将会经常在XPath表达式中使用到类。在这些情况下，需要记住由于一些被称为CSS的样式元素，你会经常看到HTML元素在其class 属性中拥有多个类。比如，在一个导航系统中，你会看到一些div标签的class 属性是"link "，而另一些是"link active "。后者是当前激活的链接，因此会表现为可见或使用一种特殊的颜色（通过CSS）高亮表示。当抓取时，你通常会对包含有特定类的元素感兴趣，具体来说，就是前面例子中的"link "和"link active "。对于这种情况，XPath的contains() 函数可以让你选择包含有指定类的所有元素。

- 选择class 属性值为"infobox "的表格中第一张图片的URL。

```
//table[@class="infobox"]//img[1]/@src
```

- 选择class 属性以"reflist "开头的div 标签中所有链接的URL。

```
//div[starts-with(@class,"reflist")]//a/@href
```

- 选择子元素包含文本"References "的元素之后的div 元素中所有链接的URL。

```
//*[text()="References"]/../following-sibling::div//a
```

请注意该表达式非常脆弱并且很容易无法使用，因为它对文档结构

做了过多假设。

- 获取页面中每张图片的URL。

```
//img/@src
```

2.2.4　预见变化

抓取时经常会指向我们无法控制的服务器页面。这就意味着如果它们的HTML以某种方式发生变化后，就会使XPath表达式失效，我们将不得不回到爬虫当中进行修正。通常情况下，这不会花费很长时间，因为这些变化一般都很小。但是，这仍然是需要避免发生的情况。一些简单的规则可以帮助我们减少表达式失效的可能性。

- 避免使用数组索引（数值）

Chrome经常会给你的表达式中包含大量常数，例如：

```
//*[@id="myid"]/div/div/div[1]/div[2]/div/div[1]/div[1]/a/img
```

这种方式非常脆弱，因为如果像广告块这样的东西在层次结构中的某个地方添加了一个额外的`div` 的话，这些数字最终将会指向不同的元

素。本案例的解决方法是尽可能接近目标的img 标签，找到一个可以使用的包含id 或者class 属性的元素，如：

```
//div[@class="thumbnail"]/a/img
```

- 类并没有那么好用

使用class 属性可以更加容易地精确定位元素，不过这些属性一般是用于通过CSS影响页面外观的，因此可能会由于网站布局的微小变更而产生变化。例如下面的class ：

```
//div[@class="thumbnail"]/a/img
```

一段时间后，可能会变成：

```
//div[@class="preview green"]/a/img
```

- 有意义的面向数据的类要比具体的或者面向布局的类更好

在前面的例子中，无论是"thumbnail "还是"green "都是我们所依赖类名的坏示例。虽然"thumbnail "比"green "确实更好一些，但是它们都不如"departure-time "。前面两个类名是用于描述布局的，

而"departure-time"更加有意义，与div标签中的内容相关。因此，在布局发生变化时，后者更可能保持有效。这可能也意味着该站的开发者非常清楚使用有意义并且一致的方式标注他们数据的好处。

- ID通常是最可靠的

通常情况下，id属性是针对一个目标的最佳选择，因为该属性既有意义又与数据相关。部分原因是JavaScript以及外部链接锚一般选择id属性以引用文档中的特定部分。例如，下面的XPath表达式非常健壮。

```
//*[@id="more_info"]//text()
```

例外情况是以编程方式生成的包含唯一标记的ID。这种情况对于抓取毫无意义。比如：

```
//[@id="order-F4982322"]
```

尽管使用了id，但上面的表达式仍然是一个非常差的XPath表达式。需要记住的是，尽管ID应该是唯一的，但是你仍然会发现很多HTML文档并没有满足这一要求。

2.3　本章小结

由于标记的质量不断提高，现在可以更加容易地创建健壮的XPath表达式，来抽取HTML文档中的数据。在本章中，你学习了HTML文档和XPath表达式的基础知识。你可以看到如何使用Google的Chrome浏览器自动获取一些XPath表达式，并将其作为我们后续优化的起点。你同样还学到了如何通过审查HTML文档，直接创建这些表达式，以及辨别XPath表达式是否健壮。现在，我们准备好运用已经学到的所有知识，在第3章中使用Scrapy编写我们的前几个爬虫。

第3章　爬虫基础

这是非常重要的一章，你可能会多次阅读本章，并且经常会在寻找解决方案时回到本章中。我们首先会介绍如何安装Scrapy，然后伴随若干示例及不同的实现，转向开发Scrapy爬虫的方法论。在开始之前，我们先来看一些重要的概念。

由于我们会快速进入有趣的代码部分，因此使用本书中代码片段的能力非常重要。当你看到如下内容时：

```
$ echo hello world

hello world
```

表示你在终端输入了echo hello word（忽略美元符号），接下来的一行或几行就是你在终端上面看到的输出。



我们将会混用“终端”、“控制台”和“命令行”这几个术语，它们在本书的背景下没有太大区别。请用Google搜索并找出如何启动你所使用的平台（Windows、OS X或其他）中的控制台。你也可以在附录A中找到详细的指引。

当你看到如下内容时：

```
>>> print 'hi'

hi
```

表示你在Python或Scrapy的shell提示符中输入了`print 'hi'`（忽略>>>）。同样地，接下来的一行或几行就是你在终端上面看到的该命令的输出。

在本书中，你还需要编辑文件。你所使用的工具很大程度上依赖于你的环境。如果你使用Vagrant（强烈推荐），可以使用电脑或笔记本

中诸如Notepad、Notepad++、Sublime Text、TextMate、Eclipse或PyCharm等编辑器。如果你有更多的Linux或UNIX使用经验，也可能更喜欢直接使用Vim或Emacs在控制台中编辑文件。这两种编辑器都很强大，不过需要一定的学习曲线。如果你是一个初学者，并且不得不在控制台中编辑某些东西，那么也可以尝试对初学者更加友好的nano编辑器。

3.1　安装Scrapy

Scrapy的安装相对来说比较简单，不过它会完全依赖于你从哪里起步。为了能够支持尽可能多的用户，本书中运行和安装Scrapy以及所有示例的“官方”方式是通过Vagrant，该软件能够让你在不考虑宿主操作系统的情况下，运行一个标准的Linux系统，在该系统中我们已经安装好所有需要用到的工具。我们将会在接下来的几小节中给出Vagrant的使用说明以及一些常用操作系统中的指引。

3.1.1　MacOS

为了更加方便地阅读本书，请按照后面给出的Vagrant使用说明操作。如果你想直接在MacOS系统中安装Scrapy，其实也很简单。只需要输入下面的命令即可。

```
$ easy_install scrapy
```

然后，一切都会为你准备好。在过程中，可能会要求你填写密码或安装Xcode，如图3.1所示。这些都没有问题，你可以放心地接受这些请求。



图3.1

3.1.2 Windows

直接在Windows系统中安装Scrapy会复杂一些，坦白来说，会有一点痛苦。而且，安装本书中所需的所有软件也需要很大程度的勇气和决心。我们已经为你做好了准备。Vagrant和Virtualbox可以在Windows 64位平台中良好运行。直接前往本章后续的相关小节，你可以很快将其安装好并运行起来。如果你必须要在Windows系统中直接安装Scrapy，请查阅本书网站（`http://scrapybook.com` ）中的资源。

3.1.3 Linux

和前面提及的两个操作系统一样，如果你想按照本书操作，那么Vagrant就是最为推荐的方式。

由于在很多场景下，你需要在Linux服务器中安装Scrapy，因此更详尽的指引可能会很有用。



确切的依赖条件经常会发生变更。本书编写时，我们安装的Scrapy版本是1.0.3，下面的内容是针对不同主流系统的操作指南。

1．Ubuntu或Debian Linux

为了在Ubuntu（使用Ubuntu 14.04 Trust Tahr 64位版本测试）或其他使用`apt`的发布版本中安装Scrapy，需要执行如下3个命令。

```
$ sudo apt-get update

$ sudo apt-get install python-pip python-lxml python-crypto python-

cssselect python-openssl python-w3lib python-twisted python-dev libxml2-
```

```
dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev

$ sudo pip install scrapy
```

上述过程需要一些编译工作，而且可能会被不时打断，不过它将会为你安装PyPI源上最新版本的Scrapy。如果你想避免某些编译工作，并且能够忍受使用稍微过时一些的版本的话，可以通过Google搜索“install Scrapy Ubuntu packages”，并跟随Scrapy官方文档的指引进行操作。

2．Red Hat或CentOS Linux

在Red Hat或其他使用yum 的发布版本中安装Scrapy相对来说比较容易。你只需按照如下3行操作即可。

```
sudo yum update

sudo yum -y install libxslt-devel pyOpenSSL python-lxml python-devel gcc
```

```
sudo easy_install scrapy
```

3.1.4　最新源码安装

只要你按照上述指引操作的话，就已经安装好了Scrapy目前所需的所有依赖。由于Scrapy是纯Python应用，因此如果你想修改其源代码或测试最新功能，可以很容易地从https://github.com/scrapy/scrapy网站中克隆其最新版本。在你的系统中安装Scrapy，只需输入如下命令。

```
$ git clone https://github.com/scrapy/scrapy.git

$ cd scrapy

$ python setup.py install
```

我猜如果你属于这类Scrapy用户，也就不需要我再提及 `virtualenv` 了。

3.1.5　升级Scrapy

Scrapy经常会升级。你会发现自己需要在很短时间内完成升级，此时可以使用`pip` 、`easy_install` 或`aptitude` 完成这项工作。

```
$ sudo pip install --upgrade Scrapy
```

或

```
$ sudo easy_install --upgrade scrapy
```

如果想降级或选择特定版本，可以通过指定版本号来完成，比如：

```
$ sudo pip install Scrapy==1.0.0
```

或

```
$ sudo easy_install scrapy==1.0.0
```

3.1.6　Vagrant：本书中运行示例的官方方式

本书中会有很多复杂但又有趣的例子，其中一些例子会用到很多服务。无论是处于初学还是进阶阶段，都可以运行本书中的这些示例，这是因为被称为Vagrant的程序可以让我们仅仅使用简单的命令就能准备好这个复杂的系统。本书中使用的系统如图3.2所示。

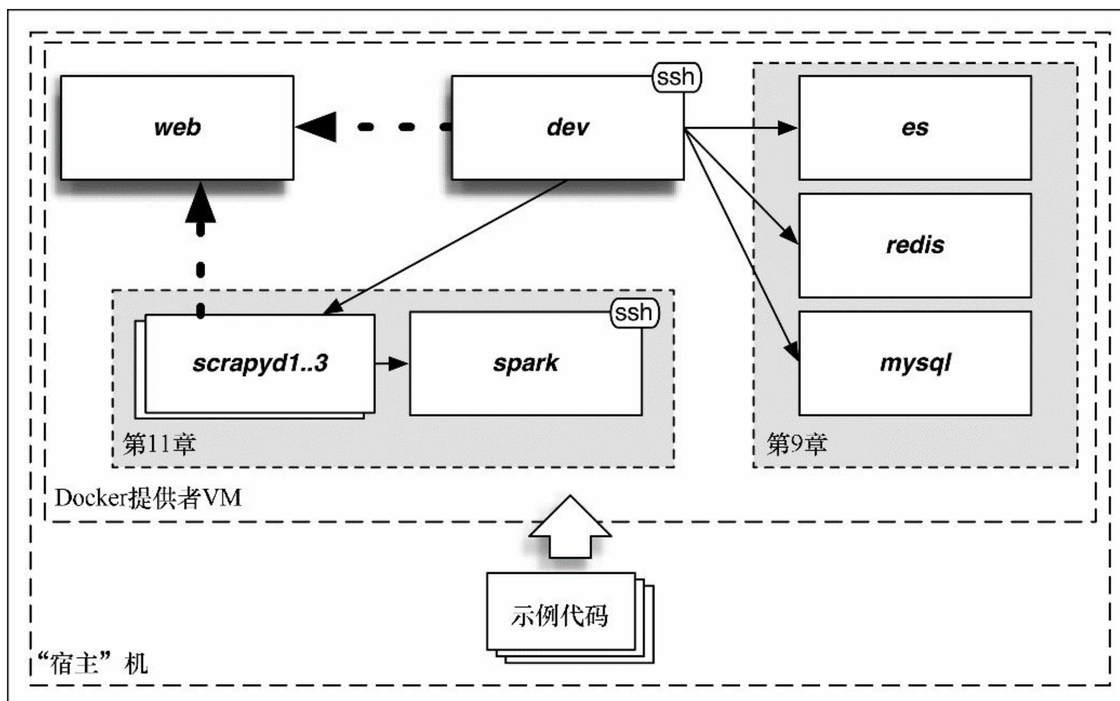


图3.2　本书使用的系统

在Vagrant的术语中，你的电脑或笔记本被称为“宿主”机。Vagrant使用宿主机运行Docker提供者VM（虚拟机）。这些技术可以让我们拥有一个隔离的系统，在其中拥有其私有网络，可以忽略宿主机的软硬件，运行本书中的示例。

大部分章节只使用了两个服务："dev"机器和"web"机器。我们登录到dev机器中运行爬虫，抓取web机器中的页面。后面的一些章节会用到更多的服务，包括数据库和大数据处理引擎。

请按照附录A的说明，在操作系统中安装Vagrant。到附录A的结尾时，你应当已经在操作系统中安装好git 和Vagrant 了。打开控制台/终端/命令提示符，现在可以按照如下操作获取本书的代码了。

```
$ git clone https://github.com/scalingexcellence/scrapybook.git
```

```
$ cd scrapybook
```

然后可以通过输入如下命令打开Vagrant系统。

```
$ vagrant up --no-parallel
```

在首次运行时将会花费一些时间，这取决于你的网络连接状况。在这之后，'vagrant up'操作将会瞬间完成。当系统运行起来之后，就可以使用如下命令登录dev虚拟机。

```
$ vagrant ssh
```

现在，你已经处于开发控制台当中，在这里可以按照本书的其他说明操作。代码已经从你的宿主机复制到dev机器当中，可以在book目录下找到这些代码。

```
$ cd book

$ ls

ch03 ch04 ch05 ch07 ch08 ch09 ch10 ch11 ...
```

打开几个控制台并执行`vagrant ssh`，可以获得多个可供操作的dev终端。可以使用`vagrant halt`关闭系统，使用`vagrant status`查看系统状态。请注意，`vagrant halt`不会关掉VM。如果出现问题，则需要打开VirtualBox然后手动关闭它，或者使用`vagrant`

`global-status`找到其id（名为"docker-provider"），然后使用`vagrant halt <ID>`停掉它。即使你处于离线状态，大部分示例仍然能够运行，这也是使用Vagrant的一个很好的副作用。

现在，我们已经正确地创建好了系统，下面就该准备学习Scrapy了。

3.2　UR^2 IM——基本抓取流程

每个网站都是不同的，如果发现某些不常见的情况，则需要一些额外的学习，或是在Scrapy的邮件列表中咨询一些问题。不过，为了知道在哪里和如何搜索，重要的是对其流程有一个整体的了解，并且清楚相关的术语。和Scrapy打交道时，你所遵循的最通用的流程是UR^2 IM流程，如图3.3所示。

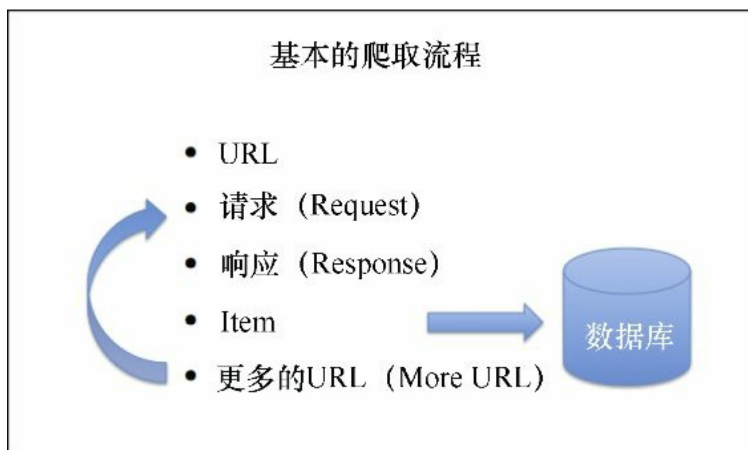


图3.3　UR^2 IM流程

3.2.1　URL

一切始于URL。你需要从准备抓取的网站中选择几个示例URL。我

将使用Gumtree分类广告网站（ https://www.gumtree.com ）作为示例进行演示。

比如，通过访问Gumtree上的伦敦房产主页（链接为 http://www.gumtree.com/flats-houses/london ），你能够找到一些房产的示例URL。可以通过右键单击分类列表，选择Copy Link Address（复制链接地址）或你浏览器中同样的功能，来复制这些链接。比如，其中一个可能类似于 https://www.gumtree.com/p/studios-bedsits-rent/split-level 。虽然可以在真实网站中使用这些URL来操作，但不幸的是，经过一段时间后，真实的Gumtree网站可能会发生变化，造成XPath表达式无法正常工作。此外，除非设置一个用户代理头，否则Gumtree不会回应你的请求。稍后我们会对此进行更进一步的讲解，不过就现在而言，如果想加载它们的某个页面，可以在scrapy shell中使用如下命令。

```
scrapy shell -s USER_AGENT="Mozilla/5.0" <your url here e.g. http://www.

gumtree.com/p/studios-bedsits-rent/...>
```

如果想要在使用scrapy shell时调试问题，可以使用`--pdb`参数启用交互式调试，以避免发生异常。例如：

```
scrapy shell --pdb https://gumtree.com
```



scrapy shell是一个非常有用的工具，能够帮助我们使用Scrapy开发。

很显然，我们并不鼓励你在学习本书内容时访问Gumtree的网站，我们也不希望本书的示例在不久之后就无法使用。此外，我们还希望即使无法连接互联网，你仍然能够开发和使用我们的示例。这就是为什么你的Vagrant开发环境中包含一个提供了类似于Gumtree网站页面的Web服务器的原因。虽然它们可能不如真实网站那么漂亮，但是从爬虫角度来说，它们其实是一样的。即便如此，我们在本章中的所有截图还是来自真实的Gumtree网站。在你Vagrant的dev机器中，可以通过`http://web:9312/`访问该Web服务器，而在你的浏览器中，可以通过`http://localhost:9312/`来访问。

在scrapy shell中打开服务器中的一个网页，并且在dev机器上输入如下内容进行操作。

```
$ scrapy shell http://web:9312/properties/property_000000.html
```

```
...
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x2d4fb10>
[s]   item       {}
[s]   request    <GET http:// web:9312/.../property_000000.html>
[s]   response   <200 http://web:9312/.../property_000000.html>
[s]   settings   <scrapy.settings.Settings object at 0x2d4fa90>
[s]   spider     <DefaultSpider 'default' at 0x3ea0bd0>
```

```
[s] Useful shortcuts:

[s]   shelp()           Shell help (print this help)

[s]   fetch(req_or_url) Fetch request (or URL) and update local...

[s]   view(response)    View response in a browser

>>>
```

我们得到了一些输出，现在可以在Python提示符下，用它来调试刚才加载的页面（一般情况下，可以使用*Ctrl* + *D* 退出）。

3.2.2 请求和响应

大家可能注意到在前面的日志中，scrapy shell本身已经为我们做了一些工作。我们给出了一个URL，然后它执行了一个默认的GET 请求，并得到了一个状态码为200 的响应。这就意味着，页面信息已经加载完毕，可以使用了。如果想要打印response.body 的前50个字符，可以按如下命令操作。

```
>>> response.body[:50]

'<!DOCTYPE html>\n<html>\n<head>\n<meta charset="UTF-8"'
```



[:50] 是什么？这是Python从文本变量（本例为response.body ）中抽取最前面50个字符（如果存在）的方式。如果你之前并不了解Python，请保持冷静，继续向前。很快，你就会熟悉并享受所有这些语法技巧了。

这是Gumtree上指定页面的HTML内容。请求和响应部分不会给我

们带来太多麻烦。不过，在很多情况下，你需要做一些工作才能保证其正确。第5章中讲到这些内容。就目前来说，我们尽量保持简单，直接进入下一部分——`Item` 。

3.2.3 Item

下一步是尝试从响应中将数据抽取到`Item` 的字段中。因为该页面的格式是HTML，因此可以使用XPath表达式进行操作。首先，让我们看一下这个页面，如图3.4所示。

图3.4　页面、感兴趣的字段及其HTML代码

在图3.4中有大量的信息，但其中大部分都是布局：logo、搜索框、按钮等。虽然这些信息都很有用，但是爬虫并不会对其产生兴趣。我们可能感兴趣的字段，比如说包括房源的标题、位置或代理商的电话号码，它们都具有对应的HTML元素，我们需要定位到这些元素，然后使

用前一节中所描述的流程抽取数据。那么，先从标题开始吧（如图3.5所示）。

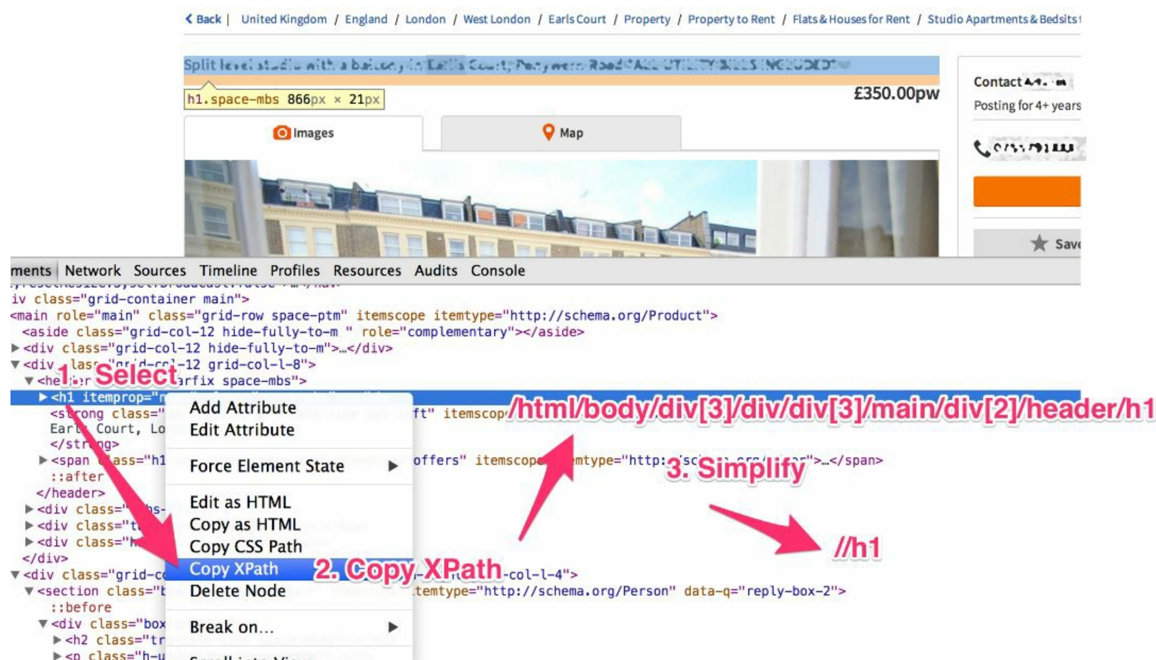


图3.5　抽取标题

右键单击页面上的标题，并选择**Inspect Element** 。这样就可以看到相应的HTML源代码了。现在，尝试通过右键单击并选择**Copy XPath** ，抽取标题的XPath表达式。你会发现Chrome浏览器给我们的XPath表达式很精确，但又十分复杂，因此该表达式是非常脆弱的。我们将对其进行一些简化，只使用最后的一部分，通过使用表达式`//h1` ，选择在页面中可以看到的任何H1 元素。尽管这种方式有些误导，因为我们并不是真的需要页面中的每一个H1 ，不过实际上这里只有标题使用了H1 ；而作为优秀的SEO实践，每个页面应当只有一个H1 元素，并且大部分网站确实是这样的。



SEO是Search Engine Optimization（搜索引擎优化）的缩写，即通过优化网站代码、内容和出入站链接的流程，实现提供给搜索引擎的最佳方式。

我们来检查下该XPath表达式能否在scrapy shell中良好运行。

```
>>> response.xpath('//h1/text()').extract()

[u'set unique family well']
```

非常好，完美工作。你应该已经注意到我在`//h1`表达式的结尾处添加了`/text()`。如果想要只抽取`H1`元素所包含的文本内容，而不是`H1`元素自身的话，就需要使用到它。我们通常都会使用`/text()`来获得文本字段。如果忽略它，就会得到整个元素的文本，包括并不需要的标记。

```
>>> response.xpath('//h1').extract()
```

```
[u'<h1 itemprop="name" class="space-mbs">set unique family well</h1>']
```

此时，我们就得到了抽取本页中第一个感兴趣的属性（标题）的代码，不过如果你观察得更仔细的话，就会发现还有一种更好更简单的方法也可以做到。

Gumtree通过微数据标记注解它们的HTML。比如，我们可以看到，在其头部有一个`itemprop="name"`的属性，如图3.6所示。非常好，这样我们就可以使用一个更简单的XPath表达式，而不再包含任何可视化元素了，此时得到的表达式为`//*[@itemprop="name"][1]/text()`。你可能会奇怪为什么我们选择了包含`itemprop="name"`的第一个元素。

▶ <h1 itemprop="name" class="space-mbs">…</h1>

图3.6　Gumtree拥有微数据标记



稍等！你是说第一个？如果你是一个经验丰富的程序员，可能已经

> 将array[1] 作为数组的第二个元素了。令人惊讶的是，XPath是从1开始的，因此array[1] 是数组的第一个元素。

我们这么做，不只是因为itemprop="name" 在许多不同的上下文中作为微数据来使用，还因为Gumtree在其页面的“你可能还喜欢……”部分为其他属性使用了嵌套的信息，以这种方式阻止我们对其轻易识别。尽管如此，这并不是一个大问题。我们只需要选择第一个，而且我们也将使用同样的方式处理其他字段。

让我们来看一下价格。价格被包含在如下的HTML结构当中。

```
<strong class="ad-price txt-xlarge txt-emphasis" itemprop="price">
£334.39pw</strong>
```

我们又一次看到了itemprop="name" 这种形式，太棒了。此时，XPath表达式将会是//*[@itemprop="price"][1]/text() 。我们来试一下吧。

```
>>> response.xpath('//*[@itemprop="price"][1]/text()').extract()

[u'\xa3334.39pw']
```

我们注意到，这里包含一些Unicode字符（英镑符号£），然后是334.39pw 的价格。这表明数据并不总是像我们希望的那样干净，所以可能还需要对其进行一些清洗的工作。比如，在本例中，我们可能需要使用一个正则表达式，以便只选择数字和点号。可以使用re() 方法做到这一要求，并使用一个简单的正则表达式替代extract() 。

```
>>> response.xpath('//*[@itemprop="price"][1]/text()').re('[.0-9]+')

[u'334.39']
```



这里使用了一个response 对象，并调用了它的xpath() 方法来抽取感兴趣的值。不过，xpath() 返回的值是什么呢？如果在一个简单的XPath表达式中，不使用.extract() 方法，将会得到如下的显示输出：

```
>>> response.xpath('.')

[<Selector xpath='.' data=u'<html>\n<head>\n<meta

charse'>]
```

xpath() 返回了网页内容预加载的Selector 对象。我们目前只使用了xpath() 方法，不过它还有另一个有用的方法：css() 。xpath() 和css() 都会返回选择器，只有当调用extract() 或re() 方法的时候，才会得到真实的文本数组。这种方式非常好用，因为这样就可以将xpath() 和css() 操作串联起来了。比如，可以使用css() 快速抽取正确的HTML元素。

```
>>> response.css('.ad-price')

[<Selector xpath=u"descendant-or-self::*[@class and

contains(concat(' ', normalize-space(@class), ' '), '

ad-price ')]" data=u'<strong class="ad-price txt-xlarge

txt-e'>]
```

请注意，在后台中css()实际上编译了一个xpath()表达式，不过我们输入的内容要比XPath自身更加简单。接下来，串联一个xpath()方法，只抽取其中的文本。

```
>>> response.css('.ad-price').xpath('text()')

[<Selector xpath='text()' data=u'\xa3334.39pw'>]
```

最后，还可以通过re() 方法，串联上正则表达式，以抽取感兴趣的值。

```
>>> response.css('.ad-price').xpath('text()').re('[.0-
9]+')
[u'334.39']
```

实际上，这个表达式与原始表达式相比，并无好坏之差。请把它当作一个引起思考的说明性示例。在本书中，我们将尽可能保持事物简单，同时也会尽可能多地使用虽然有些老旧但仍然好用的XPath。关键点是记住`xpath()`和`css()`返回的`Selector`对象是可以被串联起来的。为了获取真实值，可以使用`extract()`，也可以使用`re()`。在Scrapy的每个新版本当中，都会围绕这些类添加新的令人兴奋且高价值的功能。相关的Scrapy文档部分为`http://doc.scrapy.org/en/latest/topics/selectors.html`。该文档非常优秀，相信你可以从中找到抽取数据的最有效的方式。

描述文本的抽取也是相似的。有一个`itemprop="description"`的属性用于标示描述。其XPath表达式为`//*[@itemprop="description"][1]/text()`。相似地，住址部分使用`itemtype="http://schema.org/ Place"`注解；因此，XPath表达式为`//*[@itemtype="http://schema.org/Place"][1]/text()`。

同理，图片使用了`itemprop="image"`。因此使用`//img[@itemprop="image"][1]/@src`。这里需要注意的是，我们没有使用`/text()`，这是因为我们并不需要任何文本，而是只需要包含图片URL的`src`属性。

假设这些是我们想要抽取的全部信息，我们可以将其总结到表3.1中。

表3.1

基本字段	**XPath表达式**

title	//*[@itemprop="name"][1]/text() 示例值：[u'set unique family well']
price	//*[@itemprop="price"][1]/text() 示例值（使用re()）：[u'334.39']
description	//*[@itemprop="description"][1]/text() 示例值：[u'website court warehouse\r\npool...']
address	//*[@itemtype="http://schema.org/Place"][1]/text() 示例值：[u'Angel, London']
image_urls	//*[@itemprop="image"][1]/@src 示例值：[u'../images/i01.jpg']

现在，表3.1就变得非常重要了，因为如果我们有许多包含相似信息的网站，则很可能需要创建很多类似的爬虫，此时只需改变前面的这些表达式。此外，如果想要抓取大量网站，也可以使用这样一张表格来拆分工作量。

到目前为止，我们主要在使用HTML和XPath。接下来，我们将开始编写一些真正的Python代码。

3.3　一个Scrapy项目

到目前为止，我们只是在通过scrapy shell“小打小闹”。现在，既然

已经拥有了用于开始第一个Scrapy项目的所有必要组成部分，那么让我们按下*Ctrl + D*退出scrapy shell吧。需要注意的是，你现在输入的所有内容都将丢失。显然，我们并不希望在每次爬取某些东西的时候都要输入代码，因此一定要谨记scrapy shell只是一个可以帮助我们调试页面、XPath表达式和Scrapy对象的工具。不要花费大量时间在这里编写复杂代码，因为一旦你退出，这些代码就都会丢失。为了编写真实的Scrapy代码，我们将使用项目。下面创建一个Scrapy项目，并将其命名为"`properties`"，因为我们正在抓取的数据是房产。

```
$ scrapy startproject properties

$ cd properties

$ tree

.

├── properties
```

```
│    ├── __init__.py

│    ├── items.py

│    ├── pipelines.py

│    ├── settings.py

│    └── spiders

│         └── __init__.py

└── scrapy.cfg


2 directories, 6 files
```



提醒一下，你可以从GitHub中获得本书的全部源代码。要下载该代码，可以使用如下命令：

```
git clone https://github.com/scalingexcellence/

scrapybook
```

本章的代码在ch03 目录中，其中该示例的代码在ch03/properties 目录中。

我们可以看到这个Scrapy项目的目录结构。命令scrapy startproject properties 创建了一个以项目名命名的目录，其中包含3个我们感兴趣的文件，分别是items.py 、pipelines.py 和settings.py 。这里还有一个名为spiders 的子目录，目前为止该目录是空的。在本章中，我们将主要在items.py 文件和spiders 目录中工作。在后续的章节里，还将对设置、管道和scrapy.cfg 文件有更多探索。

3.3.1　声明item

我们使用一个文件编辑器打开items.py 文件。现在该文件中已经包含了一些模板代码，不过还需要针对用例对其进行修改。我们将重定义PropertiesItem 类，添加表3.2中总结出来的字段。

我们还会添加几个字段，我们的应用在后续会用到这些字段（这样之后就不需要再修改这个文件了）。本书后续的内容会深入解释它们。需要重点注意的一个事情是，我们声明一个字段并不意味着我们将在每个爬虫中都填充该字段，或是全部使用它。你可以随意添加任何你感觉合适的字段，因为你可以在之后更正它们。

表3.2

可计算字段	**Python**表达式

images	图像管道将会基于image_urls 自动填充该字段。可以在后续的章节中了解更多相关内容
location	我们的地理编码管道将会在后面填充该字段。可以在后续的章节中了解更多相关的内容

我们还会添加一些管理字段（见表3.3）。这些字段不是特定于某个应用程序的，而是我个人感兴趣的字段，可能会在未来帮助我调试爬虫。你可以在项目中选择其中的一些字段，当然也可以不选择。如果你仔细观察这些字段，就会明白它们可以让我清楚何地（server、url）、何时（date）、如何（spider）执行的抓取。它们还可以自动完成一些任务，比如使item失效、规划新的抓取迭代或是删除来自有问题的爬虫的item。如果你还不能理解所有的表达式，尤其是server的表达式，也不用担心。当我们进入到后面的章节时，这些都会变得越来越清楚。

表**3.3**

管理字段	**Python**表达式
url	response.url 示例值：'http://web.../property_000000. html'
project	self.settings.get('BOT_NAME') 示例值：'properties'
spider	self.name 示例值：'basic'

server	socket.gethostname() 示例值：'scrapyserver1'
date	datetime.datetime.now() 示例值：datetime.datetime(2015, 6, 25...)

给出字段列表之后，再去修改并自定义scrapy startproject 为我们创建的PropertiesItem 类，就会变得很容易。在文本编辑器中，修改properties/items.py 文件，使其包含如下内容：

```
from scrapy.item import Item, Field

class PropertiesItem(Item):
    # Primary fields
    title = Field()
    price = Field()
    description = Field()
    address = Field()
    image_urls = Field()

    # Calculated fields
    images = Field()
    location = Field()

    # Housekeeping fields
    url = Field()
    project = Field()
    spider = Field()
    server = Field()
    date = Field()
```

由于这实际上是我们在文件中编写的第一个Python代码，因此需要

重点指出的是，Python使用缩进作为其语法的一部分。在每个字段的起始部分，会有精确的4个空格或1个制表符，这一点非常重要。如果你在其中一行使用了4个空格，而在另一行使用了3个空格，就会出现语法错误。如果你在其中一行使用了4个空格，而在另一行使用了制表符，同样也会产生语法错误。这些空格在`PropertiesItem`类下，将字段声明组织到了一起。其他语言一般使用大括号（{}）或特殊的关键词（如`begin-end`）来组织代码，而Python使用空格。

3.3.2　编写爬虫

我们已经在半路上了。现在，我们需要编写爬虫。通常，我们会为每个网站或网站的一部分（如果网站非常大的话）创建一个爬虫。爬虫代码实现了完整的UR^2 IM流程，我们很快就可以看到。



什么时候使用爬虫，什么时候使用项目呢？项目是由`Item`和若干爬虫组成的。如果有很多网站，并且需要从中抽取相同类型的`Item`，比如：房产，那么所有这些网站都可以使用同一个项目，并且为每个源/网站使用一个爬虫。反之，如果要处理图书及房产这两种不同的源时，则应该使用不同的项目。

当然，可以在文本编辑器中从头开始创建一个爬虫，不过为了减少一些输入，更好的方法是使用`scrapy genspider`命令，如下所示。

```
$ scrapy genspider basic web
```

```
Created spider 'basic' using template 'basic' in module:

  properties.spiders.basic
```

现在，如果再次运行tree 命令，就会注意到与之前相比唯一的不同是在properties/spiders 目录中增加了一个新文件basic.py 。前面的命令所做的工作就是创建了一个名为"basic" 的“默认”爬虫，并且该爬虫被限制为只能爬取web 域名下的URL 。如果需要的话，可以很容易地移除这个限制，不过目前来说没有问题。爬虫使用"basic" 模板创建。你可以通过输入scrapy genspider-l 来查看其他可用的模板，然后在执行scrapy genspider 时，通过-t 参数，使用任意其他模板创建爬虫。在本章稍后的部分，我们将会看到一个示例。



Scrapy有许多子目录。我们一般假设你位于包含scrapy.cfg 文件的目录中。这是项目的“顶级”目录。现在，每当我们引用Python“包”和“模

> 块”时，它们就是以映射目录结构的方式设置的。比如，输出提到了properties.spiders.basic，就是指properties/spiders 目录中的basic.py 文件。我们早前定义的PropertiesItem 类是在properties.items 模块中，该模块对应的就是properties 目录中的items.py 文件。

如果查看properties/spiders/basic.py 文件，可以看到如下代码。

```
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_urls = (
        'http://www.web/',
    )

    def parse(self, response):
        pass
```

import 语句能够让我们使用Scrapy框架中已有的类。下面是扩展自scrapy.Spider 的BasicSpider 类的定义。通过“扩展”的方式，尽管我们实际上没有写任何代码，但是该类已经“继承”了Scrapy框架中的Spider 类的相当一部分功能。这样，就可以只额外编写少量的代码行，而获得一个完整运行的爬虫了。然后，我们可以看到一些爬虫的参数，比如它的名字以及我们允许其爬取的域名。最后是空函数parse()的定义，该函数包含了两个参数，分别是self 和response 对象。通过使用self 引用，我们就可以使用爬虫中感兴趣的功能了。而另一个对

象response，我们应该很熟悉，它就是我们在scrapy shell中使用过的response对象。



这是你的代码——你的爬虫。不要害怕修改它，你不会真的把事情搞砸的。即使在最坏的情况下，你还可以使用rmproperties/spiders/basic.py* 删除文件，然后再重新生成。尽情发挥吧！

好了，让我们开始改造吧。首先，要使用在scrapy shell中使用过的那个URL，对应地设置到start_urls参数中。然后，将使用爬虫预定义的方法log()，输出在基本字段表中总结的所有内容。修改后，properties/spiders/basic.py的代码如下所示。

```
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_urls = (
        'http://web:9312/properties/property_000000.html',
    )

    def parse(self, response):
        self.log("title: %s" % response.xpath(
            '//*[@itemprop="name"][1]/text()').extract())
        self.log("price: %s" % response.xpath(
            '//*[@itemprop="price"][1]/text()').re('[.0-9]+'))
        self.log("description: %s" % response.xpath(
            '//*[@itemprop="description"][1]/text()').extract())
        self.log("address: %s" % response.xpath(
            '//*[@itemtype="http://schema.org/'
            'Place"][1]/text()').extract())
        self.log("image_urls: %s" % response.xpath(
```

```
        '//*[@itemprop="image"][1]/@src').extract())
```



我将会不时地修改格式，以便在屏幕和纸张中都能很好地显示。这并不意味着它有什么特殊的含义。

等了这么久，终于到了运行爬虫的时候了。我们可以使用命令 scrapy crawl 以及爬虫的名称来运行爬虫。

```
$ scrapy crawl basic

INFO: Scrapy 1.0.3 started (bot: properties)

...

INFO: Spider opened
```

```
DEBUG: Crawled (200) <GET http://...000.html>

DEBUG: title: [u'set unique family well']

DEBUG: price: [u'334.39']

DEBUG: description: [u'website...']

DEBUG: address: [u'Angel, London']

DEBUG: image_urls: [u'../images/i01.jpg']

INFO: Closing spider (finished)

...
```

非常好！不要被大量的日志行吓倒。我们将会在后续的章节中更详细地研究其中的一部分，不过对于现在而言，只需要注意到所有使用XPath表达式收集到的数据确实能够通过这个简单的爬虫代码抽取出来就可以了。

让我们再来试验一下另一个命令：`scrapy parse`。它允许我们使用“最合适”的爬虫来解析参数中给定的任意URL。我不喜欢抱有侥幸心理，所以我们使用它结合--spider参数来设置爬虫。

```
$ scrapy parse --spider=basic http://web:9312/properties/property_000001.



html
```

你会看到输出和之前是相似的，只不过现在是另一套房产。



scrapy parse 同样也是一个相当方便的调试工具。在任何情况下，如果你想“认真”抓取的话，应当使用主命令scrapy crawl 。

3.3.3 填充item

我们将会对前面的代码进行少量修改，以填充PropertiesItem 。你将会看到，尽管修改非常轻微，但是会“解锁”大量的新功能。

首先，需要引入PropertiesItem 类。如前所述，它在properties 目录的items.py 文件中，也就是properties.items 模块中。我们回到properties/spiders/basic.py 文件，使用如下命令引入该模块。

```
from properties.items import PropertiesItem
```

然后需要进行实例化，并返回一个对象。这非常简单。在parse() 方法中，可以通过添加item = PropertiesItem() 语句创建一个新的item，然后可以按如下方式为其字段分配表达式。

```
item['title'] =
response.xpath('//*[@itemprop="name"][1]/text()').extract()
```

最后，使用`return item`返回item。最新版的`properties/spiders/basic.py`代码如下所示。

```
import scrapy
from properties.items import PropertiesItem

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]
    start_urls = (
        'http://web:9312/properties/property_000000.html',
    )

    def parse(self, response):
        item = PropertiesItem()
        item['title'] = response.xpath(
            '//*[@itemprop="name"][1]/text()').extract()
        item['price'] = response.xpath(
            '//*[@itemprop="price"][1]/text()').re('[.0-9]+')
        item['description'] = response.xpath(
            '//*[@itemprop="description"][1]/text()').extract()
        item['address'] = response.xpath(
            '//*[@itemtype="http://schema.org/'
            'Place"][1]/text()').extract()
        item['image_urls'] = response.xpath(
            '//*[@itemprop="image"][1]/@src').extract()
        return item
```

现在，如果你再像之前那样运行`scrapy crawl basic`，就会发现一个非常小但很重要的区别。我们不再在日志中记录抓取值（所以没有包含字段值的`DEBUG:`行了），而是看到如下的输出行。

```
DEBUG: Scraped from <200
http://...000.html>
  {'address': [u'Angel, London'],
   'description': [u'website ... offered'],
   'image_urls': [u'../images/i01.jpg'],
   'price': [u'334.39'],
```

```
 'title': [u'set unique family well']}
```

这是从本页面抓取得到的`PropertiesItem`。非常好，因为Scrapy是围绕着`Items`的概念构建的，也就是说你现在可以使用后续章节中介绍的管道，对其进行过滤和丰富了，并且可以通过“Feed exports”将其以不同的格式导出存储到不同的地方。

3.3.4 保存文件

请尝试如下爬取示例。

```
$ scrapy crawl basic -o items.json

$ cat items.json

[{"price": ["334.39"], "address": ["Angel, London"], "description":

["website court ... offered"], "image_urls": ["../images/i01.jpg"],

"title": ["set unique family well"]}]
```

```
$ scrapy crawl basic -o items.jl

$ cat items.jl

{"price": ["334.39"], "address": ["Angel, London"], "description":

["website court ... offered"], "image_urls": ["../images/i01.jpg"],

"title": ["set unique family well"]}

$ scrapy crawl basic -o items.csv

$ cat items.csv
```

```
description,title,url,price,spider,image_urls...

"...offered",set unique family well,,334.39,,../images/i01.jpg

$ scrapy crawl basic -o items.xml

$ cat items.xml

<?xml version="1.0" encoding="utf-8"?>

<items><item><price><value>334.39</value></price>...</item></items>
```

我们不需要编写任何额外的代码，就可以保存为这些不同的格式。Scrapy在幕后识别你想要输出的文件扩展名，并以适当的格式输出到文

件中。前面的格式覆盖了一些最常见的用例。CSV和XML文件非常流行，因为类似微软Excel的电子表格程序可以直接打开它们。JSON文件在网上非常流行，原因是它们富有表现力而且与JavaScript的关系相当密切。JSON与JSON行（JSON Line）格式的轻微不同是，`.json` 文件是在一个大数组中存储JSON对象的。这就意味着如果你有一个1GB的文件，你可能不得不在使用典型的解析器解析之前，将其全部存入内存当中。而`.jl` 文件则是每行包含一个JSON对象，所以它们可以被更高效地读取。

将你生成的文件保存到文件系统之外的地方也很容易。比如，通过使用如下命令，Scrapy可以自动将文件上传到FTP或S3存储桶中。

```
$ scrapy crawl basic -o "ftp://user:pass@ftp.scrapybook.com/items.json "

$ scrapy crawl basic -o "s3://aws_key:aws_secret@scrapybook/items.json"
```

需要注意的是，除非凭证和URL都更新为与有效的主机/S3提供商相匹配，否则该示例无法工作。



我的MySQL驱动在哪里？起初，我也对Scrapy缺少针对MySQL或其他数据库的内置支持感到惊讶。而实际上，没有什么是内置的，这与Scrapy的思考方式是完全违背的。Scrapy的目标是快速和可扩展。它使用了很少的CPU，以及尽可能高的入站带宽。从性能的角度来看，将数据插入到大部分关系型数据库将会是一场灾难。当需要将item插入到数据库时，必须将其先存储到文件当中，然后再使用批量加载机制导入它们。在第9章中，我们将会看到多种高效的方式，用来将独立的item导入到数据库中。

这里需要注意的另一件事是，如果你现在尝试使用scrapy parse，它会向你显示已经抓取的item，以及你的爬取生成的新请求（本例中没有）。

```
$ scrapy parse --spider=basic http://web:9312/properties/property_000001.

html

INFO: Scrapy 1.0.3 started (bot: properties)

...
```

```
INFO: Spider closed (finished)

>>> STATUS DEPTH LEVEL 1 <<<

# Scraped Items ------------------------------------------------

[{'address': [u'Plaistow, London'],

  'description': [u'features'],

  'image_urls': [u'../images/i02.jpg'],

  'price': [u'388.03'],

  'title': [u'belsize marylebone...deal']}]
```

```
# Requests -----------------------------------------------

[]
```

在调试给出意料之外的结果的URL时，你会更加感激scrapy parse 。

3.3.5　清理——item装载器与管理字段

恭喜，你在创建基础爬虫方面做得不错！下面让我们做得更专业一些吧。

首先，我们使用一个强大的工具类——ItemLoader ，以替代那些杂乱的extract() 和xpath() 操作。通过使用该类，我们的parse() 方法会按如下进行代码变更。

```
def parse(self, response):
    l = ItemLoader(item=PropertiesItem(), response=response)

    l.add_xpath('title', '//*[@itemprop="name"][1]/text()')
    l.add_xpath('price', './/*[@itemprop="price"]'
          '[1]/text()', re='[,.0-9]+')
```

```
    l.add_xpath('description', '//*[@itemprop="description"]'
          '[1]/text()')
    l.add_xpath('address', '//*[@itemtype='
          '"http://schema.org/Place"][1]/text()')
    l.add_xpath('image_urls', '//*[@itemprop="image"][1]/@src')

    return l.load_item()
```

好多了，是不是？不过，这种写法并不只是在视觉上更加舒适，它还非常明确地声明了我们意图去做的事情，而不会将其与实现细节混淆起来。这就使得代码具有更好的可维护性以及自描述性。

`ItemLoader`提供了许多有趣的结合数据及对数据进行格式化和清洗的方式。请注意，此类功能的开发非常活跃，因此请查阅Scrapy优秀的官方文档来发现使用它们的更高效的方式，文档地址为`http://doc.scrapy.org/en/latest/topics/loaders.html`。`Itemloaders`通过不同的处理类传递XPath/CSS表达式的值。处理器是一个快速而又简单的函数。处理器的一个例子是`Join()`。假设你已经使用类似`//p`的XPath表达式选取了很多个段落，该处理器可以将这些段落结合成一个条目。另一个非常有意思的处理器是`MapCompose()`。通过该处理器，你可以使用任意Python函数或Python函数链，以实现复杂的功能。比如，`MapCompose(float)`可以将字符串数据转换为数值，而`MapCompose(Unicode.strip, Unicode.title)`可以删除多余的空白符，并将字符串格式化为每个单词均为首字母大写的样式。让我们看一些处理器的例子，如表3.4所示。

表**3.4**

处　理　器	功　　能
Join()	把多个结果连接在一起
MapCompose(unicode.strip)	去除首尾的空白符
MapCompose(unicode.strip, unicode. title)	与MapCompose(unicode.strip) 相同，不过还会使结果按照标题格式
MapCompose(float)	将字符串转为数值
MapCompose(lambda i: i.replace (',', ''), float)	将字符串转为数值，并忽略可能存在的','字符
MapCompose(lambda i: urlparse. urljoin (response.url, i))	以response.url 为基础，将URL相对路径转换为URL绝对路径

你可以使用任何Python表达式作为处理器。可以看到，我们可以很容易地将它们一个接一个地连接起来，比如，我们前面给出的去除首尾空白符以及标题化的例子。unicode.strip() 和unicode.title() 在某种意义上来说比较简单，它们只有一个参数，并且也只有一个返回结果。我们可以在MapCompose 处理器中直接使用它们。而另一些函数，像replace() 或urljoin()，就会稍微有点复杂，它们需要多个参数。对于这种情况，我们可以使用Python的“lambda表达式”。lambda表达式是一种简洁的函数。比如下面这个简洁的lambda表达式。

```
myFunction = lambda i: i.replace(',', '')
```

可以代替：

```
def myFunction(i):
    return i.replace(',', '')
```

通过使用lambda，我们将类似`replace()`和`urljoin()`这样的函数包装在只有一个参数及一个返回结果的函数中。为了能够更好地理解表3.4中的处理器，下面看几个使用处理器的例子。使用`scrapy shell`打开任意URL，然后尝试如下操作。

```
>>> from scrapy.loader.processors import MapCompose, Join

>>> Join()(['hi','John'])

u'hi John'

>>> MapCompose(unicode.strip)([u' I',u' am\n'])
```

```
[u'I', u'am']

>>> MapCompose(unicode.strip, unicode.title)([u'nIce cODe'])

[u'Nice Code']

>>> MapCompose(float)(['3.14'])

[3.14]

>>> MapCompose(lambda i: i.replace(',', ''), float)(['1,400.23'])

[1400.23]

>>> import urlparse
```

```
>>>  mc = MapCompose(lambda i: urlparse.urljoin('http://my.com/test/abc',

i))

>>> mc(['example.html#check'])

['http://my.com/test/example.html#check']

>>> mc(['http://absolute/url#help'])

['http://absolute/url#help']
```

这里要解决的关键问题是，处理器只是一些简单小巧的功能，用来对我们的XPath/CSS结果进行后置处理。现在，在爬虫中使用几个这样的处理器，并按照我们想要的方式输出。

```
def parse(self, response):
    l.add_xpath('title', '//*[@itemprop="name"][1]/text()',
                MapCompose(unicode.strip, unicode.title))
    l.add_xpath('price', './/*[@itemprop="price"][1]/text()',
                MapCompose(lambda i: i.replace(',', ''), float),
                re='[,.0-9]+')
    l.add_xpath('description', '//*[@itemprop="description"]'
                '[1]/text()', MapCompose(unicode.strip), Join())
    l.add_xpath('address',
                '//*[@itemtype="http://schema.org/Place"][1]/text()',
                MapCompose(unicode.strip))
    l.add_xpath('image_urls', '//*[@itemprop="image"][1]/@src',
                MapCompose(
                lambda i: urlparse.urljoin(response.url, i)))
```

完整列表将会在本章后续部分给出。当你使用我们目前开发的代码运行`scrapy crawl basic`时，可以得到更加整洁的输出值。

```
'price': [334.39],

'title': [u'Set Unique Family Well']
```

最后，我们可以通过使用add_value() 方法，添加Python计算得出的单个值（而不是XPath/CSS表达式）。我们可以用该方法设置“管理字段”，比如URL、爬虫名称、时间戳等。我们还可以直接使用管理字段表中总结出来的表达式，如下所示。

```
l.add_value('url', response.url)
l.add_value('project', self.settings.get('BOT_NAME'))
l.add_value('spider', self.name)
l.add_value('server', socket.gethostname())
l.add_value('date', datetime.datetime.now())
```

为了能够使用其中的某些函数，请记得引入datetime 和socket 模块。

好了！我们现在已经得到了非常不错的Item 。此刻，你的第一感觉可能是所做的这些都很复杂，你可能想要知道这些工作是不是值得付出努力。答案当然是值得的——这是因为，这就是你为了从页面抽取数据并将其存储到Item 中几乎所有需要知道的东西。如果你从零开始编写，或者使用其他语言，该代码通常都会非常难看，而且很快就会变得不可维护。而使用Scrapy时，只需要仅仅25行代码。该代码十分简洁，用于表明意图，而不是实现细节。你清楚地知道每一行代码都在做什么，并且它可以很容易地修改、复用及维护。

你可能产生的另一个感觉是所有的处理器以及ItemLoader 并不值

得去努力。如果你是一个经验丰富的Python开发者，可能会觉得有些不舒服，因为你必须去学习新的类，来实现通常使用字符串操作、lambda表达式以及列表推导式就可以完成的操作。不过，这只是`ItemLoader`及其功能的简要概述。如果你更加深入地了解它，就不会再回头了。`ItemLoader`和处理器是基于编写并支持了成千上万个爬虫的人们的抓取需求而开发的工具包。如果你准备开发多个爬虫的话，就非常值得去学习使用它们。

3.3.6 创建contract

contract有点像为爬虫设计的单元测试。它可以让你快速知道哪里有运行异常。例如，假设你在几个星期之前编写了一个抓取程序，其中包含几个爬虫，今天想要检查一下这些爬虫是否仍然能够正常工作，就可以使用这种方式。contract包含在紧挨着函数名的注释（即文档字符串）中，并且以@开头。下面来看几个contract的例子。

```
def parse(self, response):
    """ This function parses a property page.

    @url http://web:9312/properties/property_000000.html
    @returns items 1
    @scrapes title price description address image_urls
    @scrapes url project spider server date
    """
```

上述代码的含义是，检查该URL，并找到我列出的字段中有值的一个Item。现在，当你运行`scrapy check`时，就会去检查contract是否能够满足。

```
$ scrapy check basic

----------------------------------------------------------------

Ran 3 contracts in 1.640s

OK
```

如果将url 字段留空（通过注释掉该行来设置），你会得到一个失败描述。

```
FAIL: [basic] parse (@scrapes post-hook)

------------------------------------------------------------------
```

```
ContractFail: 'url' field is missing
```

contract失败的原因可能是爬虫代码无法运行，或者是你要检查的URL的XPath表达式已经过时了。虽然结果并不详尽，但它是抵御坏代码的第一道灵巧的防线。

综合上面的内容，下面给出我们的第一个基础爬虫的代码。

```
from scrapy.loader.processors import MapCompose, Join
from scrapy.loader import ItemLoader
from properties.items import PropertiesItem
import datetime
import urlparse
import socket
import scrapy

class BasicSpider(scrapy.Spider):
    name = "basic"
    allowed_domains = ["web"]

    # Start on a property page
    start_urls = (
        'http://web:9312/properties/property_000000.html',
    )

    def parse(self, response):
        """ This function parses a property page.
        @url http://web:9312/properties/property_000000.html
        @returns items 1
        @scrapes title price description address image_urls
        @scrapes url project spider server date
```

```
        """
        # Create the loader using the response
        l = ItemLoader(item=PropertiesItem(), response=response)

        # Load fields using XPath expressions
        l.add_xpath('title', '//*[@itemprop="name"][1]/text()',
                    MapCompose(unicode.strip, unicode.title))
        l.add_xpath('price', './/*[@itemprop="price"][1]/text()',
                    MapCompose(lambda i: i.replace(',', ''),
                    float),
                    re='[,.0-9]+')
        l.add_xpath('description', '//*[@itemprop="description"]'
                    '[1]/text()',
                    MapCompose(unicode.strip), Join())
        l.add_xpath('address',
                    '//*[@itemtype="http://schema.org/Place"]'
                    '[1]/text()',
                    MapCompose(unicode.strip))
        l.add_xpath('image_urls', '//*[@itemprop="image"]'
                    '[1]/@src', MapCompose(
                    lambda i: urlparse.urljoin(response.url, i)))

        # Housekeeping fields
        l.add_value('url', response.url)
        l.add_value('project', self.settings.get('BOT_NAME'))
        l.add_value('spider', self.name)
        l.add_value('server', socket.gethostname())
        l.add_value('date', datetime.datetime.now())

        return l.load_item()
```

3.4 抽取更多的URL

到目前为止，我们使用的只是设置在爬虫的`start_urls`属性中的单一URL。而该属性实际为一个元组，我们可以硬编码写入更多的URL，如下所示。

```
start_urls = (
```

```
    'http://web:9312/properties/property_000000.html',
    'http://web:9312/properties/property_000001.html',
    'http://web:9312/properties/property_000002.html',
)
```

这种写法可能不会让你太激动。不过，我们还可以使用文件作为URL的源，写法如下所示。

```
start_urls = [i.strip() for i in
open('todo.urls.txt').readlines()]
```

这种写法其实也不那么令人激动，但它确实管用。更经常发生的情况是感兴趣的网站中包含一些索引页及房源页。比如，Gumtree就包含了如图3.7所示的索引页，其地址为http://www.gumtree.com/flats-houses/london 。

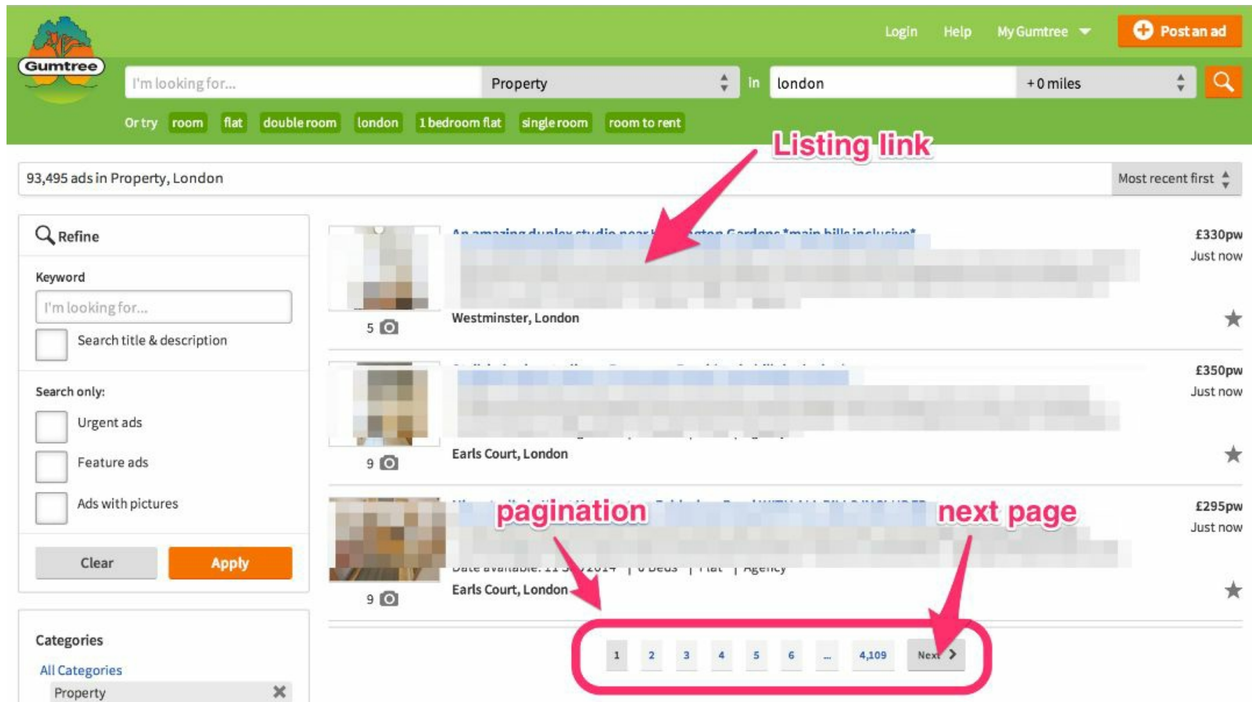


图3.7　Gumtree的索引页

一个典型的索引页会包含许多到房源页面的链接，以及一个能够让你从一个索引页前往另一个索引页的分页系统。

因此，一个典型的爬虫会向两个方向移动（见图3.8）：



图3.8　向两个方向移动的典型爬虫

- 横向——从一个索引页到另一个索引页；
- 纵向——从一个索引页到房源页并抽取`Item` 。

在本书中，我们将前者称为水平爬取 ，因为这种情况下是在同一层级下爬取页面（比如索引页）；而将后者称为垂直爬取 ，因为该方式是从一个更高的层级（比如索引页）到一个更低的层级（比如房源页）。

实际上，它比听起来更加容易。我们所有需要做的事情就是再增加两个XPath表达式。对于第一个表达式，右键单击**Next Page** 按钮，可以注意到URL包含在一个链接中，而该链接又是在一个拥有类名`next` 的

li 标签内，如图3.9所示。因此，我们只需使用一个实用的XPath表达式//*[contains(@class,"next")]//@href，就可以完美运行了。

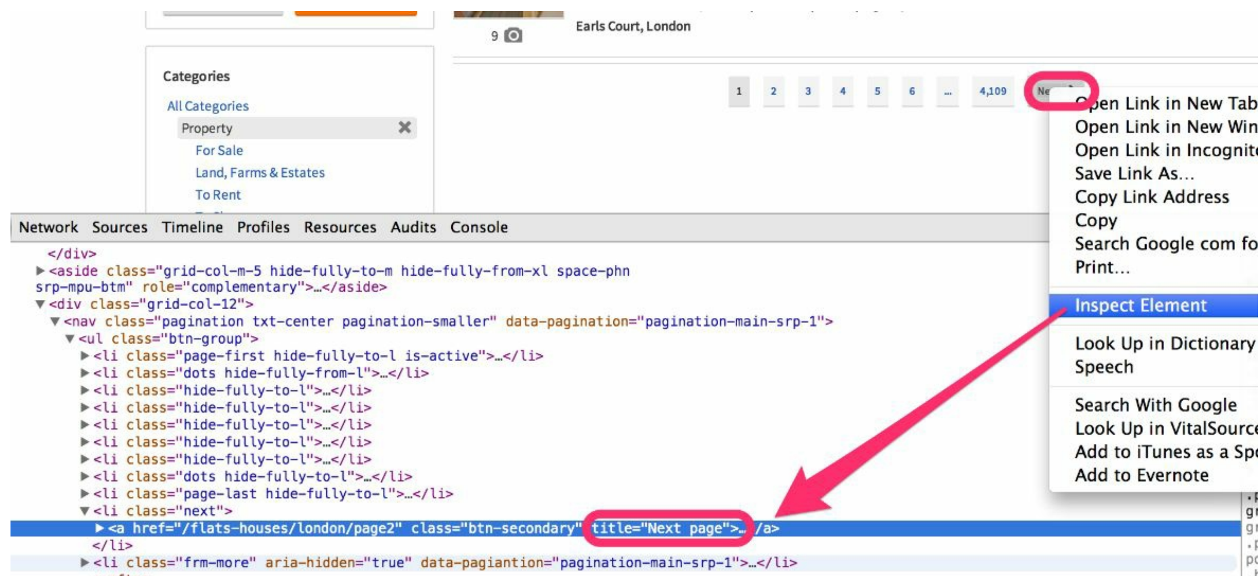


图3.9　查找下一个索引页URL的XPath表达式

对于第二个表达式，右键单击页面中的列表标题，并选择**Inspect Element**，如图3.10所示。

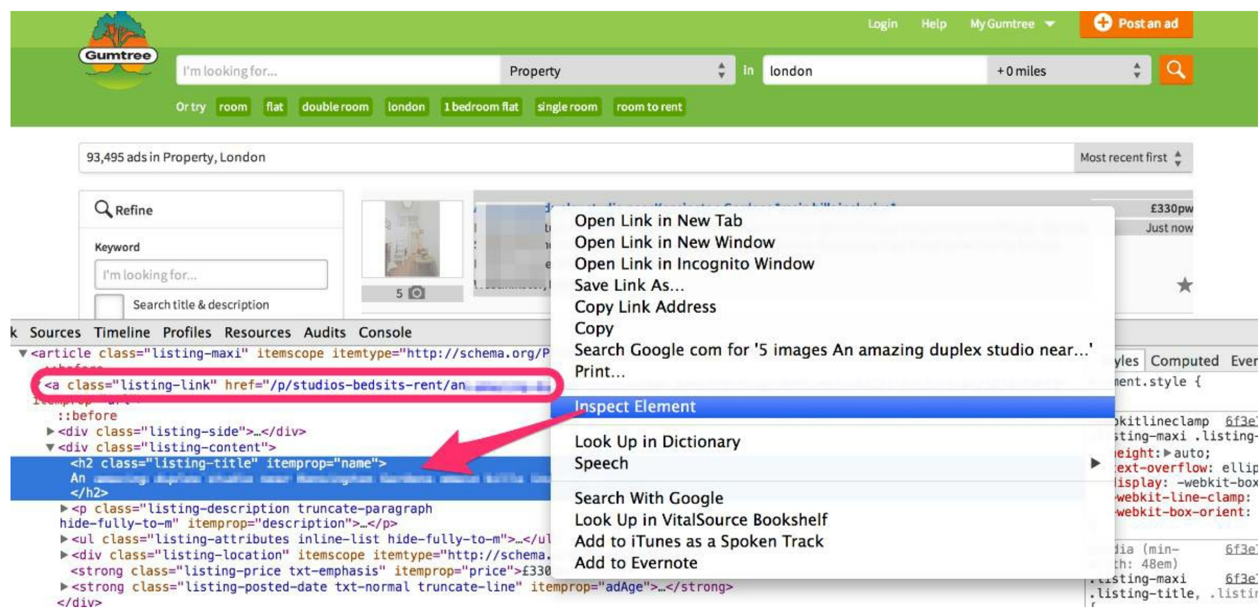


图3.10　查找列表页URL的XPath表达式

请注意，URL中包含我们感兴趣的`itemprop="url"`属性。因此，表达式`//*[@itemprop="url"]/@href`就可以正常运行。现在，打开一个scrapy shell来确认这两个表达式是否有效：

```
$ scrapy shell http://web:9312/properties/index_00000.html

>>> urls = response.xpath('//*[contains(@class,"next")]//@href').extract()

>>> urls

[u'index_00001.html']

>>> import urlparse

>>> [urlparse.urljoin(response.url, i) for i in urls]

[u'http://web:9312/scrapybook/properties/index_00001.html']
```

```
>>> urls = response.xpath('//*[@itemprop="url"]/@href').extract()

>>> urls

[u'property_000000.html', ... u'property_000029.html']

>>> len(urls)

30

>>> [urlparse.urljoin(response.url, i) for i in urls]

[u'http://..._000000.html', ... /property_000029.html']
```

```
```

非常好！可以看到，通过使用之前已经学习的内容及这两个XPath表达式，我们已经能够按照自身需求使用水平抓取和垂直抓取的方式抽取URL了。

3.4.1　使用爬虫实现双向爬取

我们将之前的爬虫拷贝到一个新文件中，并命名为manual.py 。

```
$ ls

properties scrapy.cfg

$ cp properties/spiders/basic.py properties/spiders/manual.py
```

在properties/spiders/manual.py 文件中，通过添加from scrapy.http import Request 语句引入Request 模块，将爬虫的

name 参数改为'manual'，修改start_urls 以使用第一个索引页，并将parse() 方法重命名为parse_item()。好了！现在开始编写一个新的parse() 方法，来实现水平和垂直两种抓取方式。

```
def parse(self, response):
    # Get the next index URLs and yield Requests
    next_selector = response.xpath('//*[contains(@class,'
                                   '"next")]//@href')
    for url in next_selector.extract():
        yield Request(urlparse.urljoin(response.url, url))

    # Get item URLs and yield Requests
    item_selector = response.xpath('//*[@itemprop="url"]/@href')
    for url in item_selector.extract():
        yield Request(urlparse.urljoin(response.url, url),
                      callback=self.parse_item)
```



你可能已经注意到了前面例子中的yield 语句。yield 与return 在某种意义上来说有些相似，都是将返回值提供给调用者。不过，和return 不同的是，yield 不会退出函数，而是继续执行for 循环。从功能上来说，前面的例子与下面的代码大体相当：

```
next_requests = []
for url in...
    next_requests.append(Request(...))
for url in...
    next_requests.append(Request(...))
return next_requests
```

yield 是Python“魔法”的一部分，它可以使日常的高效编程工作更加轻松。

我们现在已经准备好运行该爬虫了。不过如果让该爬虫以当前的方式运行的话，则会抓取网站完整的5万个页面。为了避免运行时间过长，可以通过命令行参数：-s CLOSESPIDER_ITEMCOUNT=90，告知爬虫在爬取指定数量（如90个）的Item后停止运行（更多细节参见第7章）。现在，我们可以运行了。

```
$ scrapy crawl manual -s CLOSESPIDER_ITEMCOUNT=90

INFO: Scrapy 1.0.3 started (bot: properties)

...

DEBUG: Crawled (200) <...index_00000.html> (referer: None)
```

```
DEBUG: Crawled (200) <...property_000029.html> (referer: ...index_00000.
html)
DEBUG: Scraped from <200 ...property_000029.html>
  {'address': [u'Clapham, London'],
  'date': [datetime.datetime(2015, 10, 4, 21, 25, 22, 801098)],
  'description': [u'situated camden facilities corner'],
  'image_urls': [u'http://web:9312/images/i10.jpg'],
  'price': [223.88],
```

```
  'project': ['properties'],

  'server': ['scrapyserver1'],

  'spider': ['manual'],

  'title': [u'Portered Mile'],

  'url': ['http://.../property_000029.html']}

DEBUG: Crawled (200) <...property_000028.html> (referer: ...index_00000.

html)

...
```

```
DEBUG: Crawled (200) <...index_00001.html> (referer: ...)
DEBUG: Crawled (200) <...property_000059.html> (referer: ...)
...
INFO: Dumping Scrapy stats: ...
   'downloader/request_count': 94, ...
   'item_scraped_count': 90,
```

如果仔细查看前面的输出，就会发现我们同时获得了水平抓取和垂

直抓取的结果。第一个index_00000.html 读取后，派生出了许多请求。当它们执行时，调试信息通过referer URL指出是谁发起的请求。比如，可以看到，property_000029.html 、property_000028.html…… 及index_00001.html 都有相同的referer（index_00000.html）。而property_000059.html 及其他请求则是以index_00001.html 为referer 的，并且该过程还在持续。

从该示例中还可以观察到，Scrapy在处理请求时使用的是后入先出（**LIFO**）策略（即深度优先爬取）。用户提交的最后一个请求会被首先处理。在大多数情况下，这种默认的方式非常方便。比如，我们想要在移动到下一个索引页之前处理每一个房源页时。否则，我们将会填充一个包含待爬取房源页URL的巨大队列，无谓地消耗内存。另外，在许多情况中，你可能需要辅助的请求来完成单个请求，我们将会在后面的章节中遇到这种情况。你需要这些辅助的请求能够尽快完成，以腾出资源，并且让被抓取的Item能够稳定流动。

我们可以通过设置Request() 的优先级参数修改默认顺序，大于0表示高于默认的优先级，小于0表示低于默认的优先级。通常来说，Scrapy的调度器会首先执行高优先级的请求，不过不要花费太多时间来考虑具体的哪个请求应该被首先执行。很可能在你的应用中，不会使用超过1个或2个请求优先级。此外还需要注意的是，URL还会被执行去重操作，这在大部分时候也是我们想要的功能。不过如果我们需要多次执行同一个URL的请求，可以设置dont_filter_Request() 参数为true。

3.4.2 使用CrawlSpider实现双向爬取

如果感觉上面的双向爬取有些冗长，则说明你确实发现了关键问题。Scrapy尝试简化所有此类通用情况，以使其编码更加简单。最简单的实现同样结果的方式是使用CrawlSpider，这是一个能够更容易地实现这种爬取的类。为了实现它，我们需要使用genspider命令，并设置-t crawl参数，以使用crawl爬虫模板创建一个爬虫。

```
$ scrapy genspider -t crawl easy web

Created spider 'crawl' using template 'crawl' in module:

  properties.spiders.easy
```

现在，文件properties/spiders/easy.py包含如下内容。

```
...
class EasySpider(CrawlSpider):
    name = 'easy'
    allowed_domains = ['web']
    start_urls = ['http://www.web/']

    rules = (
        Rule(LinkExtractor(allow=r'Items/'),
```

```
callback='parse_item', follow=True),
    )

    def parse_item(self, response):
        ...
```

当你阅读这段自动生成的代码时，会发现它和之前的爬虫有些相似，不过在此处的类声明中，会发现爬虫是继承自`CrawlSpider`，而不再是`Spider`。`CrawlSpider`提供了一个使用`rules`变量实现的`parse()`方法，这与我们之前例子中手工实现的功能一致。



你可能会感到疑惑，为什么我首先给出了手工实现的版本，而不是直接给出捷径。这是因为你在手工实现的示例中，学会了使用回调的`yield`方式的请求，这是一个非常有用和基础的技术，我们将会在后续的章节中不断使用它，因此理解该内容非常值得。

现在，我们要把`start_urls`设置成第一个索引页，并且用我们之前的实现替换预定义的`parse_item()`方法。这次我们将不再需要实现任何`parse()`方法。我们将预定义的`rules`变量替换为两条规则，即水平抓取和垂直抓取。

```
rules = (
Rule(LinkExtractor(restrict_xpaths='//*[contains(@class,"next")]')),
Rule(LinkExtractor(restrict_xpaths='//*[@itemprop="url"]'),
         callback='parse_item')
)
```

这两条规则使用的是和我们之前手工实现的示例中相同的XPath表达式，不过这里没有了a 或href 的限制。顾名思义，LinkExtractor 正是专门用于抽取链接的，因此在默认情况下，它们会去查找a （及area ）href 属性。你可以通过设置LinkExtractor() 的tags 和attrs 参数来进行自定义。需要注意的是，回调参数目前是包含回调方法名称的字符串（比如'parse_item' ），而不是方法引用，如Request(self.parse_item) 。最后，除非设置了callback 参数，否则Rule 将跟踪已经抽取的URL，也就是说它将会扫描目标页面以获取额外的链接并跟踪它们。如果设置了callback ，Rule 将不会跟踪目标页面的链接。如果你希望它跟踪链接，应当在callback 方法中使用return 或yield 返回它们，或者将Rule() 的follow 参数设置为true 。当你的房源页既包含Item 又包含其他有用的导航链接时，该功能可能会非常有用。

运行该爬虫，可以得到和手工实现的爬虫相同的结果，不过现在使用的是一个更加简单的源代码。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90
```

3.5　本章小结

本章可能是大家开始学习Scrapy时最重要的一章。你刚刚学习了开发爬虫最基本的方法：UR^2 IM。你学会了如何自定义适合需求的`Item`，使用`ItemLoader`、XPath表达式和处理器加载`Item`，以及如何对`Request`使用`yield`操作。我们使用`Request`横向到达不同的索引页，纵向到达房源页并抽取`Item`。最后，我们看到了如何使用`CrawlSpider`和`Rule`，以很少的代码行创建非常强大的爬虫。如果你想要更深入地理解这些概念，请尽可能多地阅读本章，当然，也可以在你开发自己的爬虫时使用本章作为参考。

我们刚刚从网站中得到了一些信息。为什么它这么重要呢？我想答案会在下一章中变得明朗起来，在下一章中，通过简单的几页内容，我们将会开发一个简单的手机应用，并使用Scrapy填充其中的数据。我想，结果会令大家印象深刻。

第4章　从Scrapy到移动应用

我能够听到人们的尖叫声：“Appery.io是什么，一个手机应用的专用平台，它和Scrapy有什么关系？”那么，眼见为实吧。你可能还会对几年前在Excel电子表格上给某个人（朋友、管理者或者客户）展示数据时的场景印象深刻。不过现如今，除非你的听众都十分老练，否则他们的期望很可能会有所不同。在接下来的几页里，你将看到一个简单的手机应用，这是一个只需几次单击就能够创建出来的最小可视化产品，其目的是向利益相关者传达抽取所得数据的力量，并回到生态系统中，以源网站网络流量的形式展示它能够带来的价值。

我将尽量保持简短的启发式示例，在这里它们将展示如何充分利用你的数据。只有当你有一个具体的应用用于消费数据时，才可以安全地略过本章。本章将会向你展示如何以当下最流行的方式——手机应用，向公众展示你的数据。

4.1　选择手机应用框架

借助于适当的工具向手机应用提供数据将是非常容易的事情。目前有许多优秀的跨平台手机应用开发框架，如PhoneGap、使用Appcelerator云服务的Appcelerator、jQuery Mobile和Sencha Touch。

本章将使用Appery.io，因为它可以让我们使用PhoneGap和jQuery Mobile快速创建iOS、Android、Windows Phone以及HTML5手机应用。

我和Scrapy都与Appery.io无任何利益关联。我会鼓励你独立进行调研，看看除了本章中提出的功能外，它是否也能符合你的需求。请注意这是一个付费服务，你可以有14天的试用期，不过在我看来，它可以让人无需动脑就能快速开发出原型，尤其是对于那些不是网络专家的人来说，为此付费是值得的。我选择该服务的主要原因是它既能提供手机应用，也能提供后端服务，也就是说我们不需要再去配置数据库、编写REST API或为服务端及手机应用使用其他一些语言。你将看到，我们一行代码都不用去编写！我们将会使用它们的在线工具；在任何时候，你都可以下载该应用，并作为PhoneGap项目，使用PhoneGap的所有功能。

在本章中，你需要接入互联网连接，以便使用Appery.io。同时，还需要注意的是该网站的布局可能在未来会有所变化。请将我们的截屏作为参考，而不要在发现该网站外观不同时感到惊讶。

4.2　创建数据库和集合

第一步是通过单击Appery.io网站上的**Sign-Up** 按钮并选取免费方案，来注册免费的Appery.io方案。你需要提供用户名、邮箱地址以及密码，然后就会创建好新账户了。等待几秒钟后，账户完成激活。然后就可以登录到Appery.io的仪表盘了。现在，开始准备创建新的数据库以及集合，如图4.1所示。

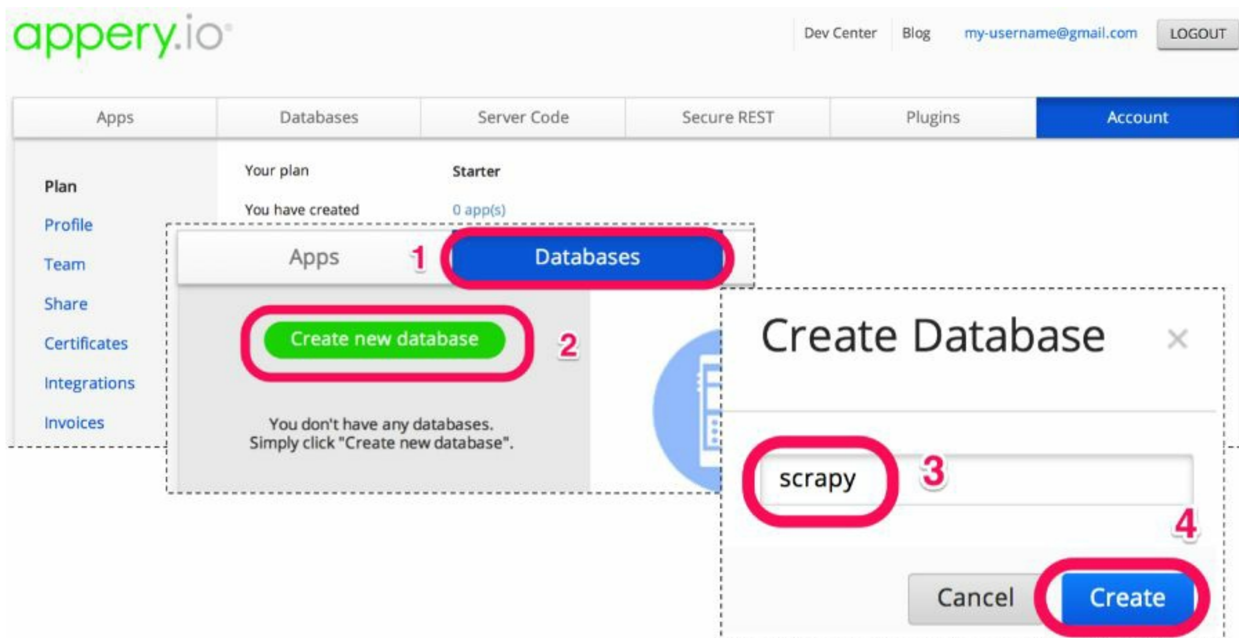


图4.1　使用Appery.io创建新数据库及集合

为了完成该操作，需要按照如下步骤执行。

1．单击**Databases** 选项卡（1）。

2．然后单击绿色的**Create new database** （2）按钮。将新数据库命名为scrapy （3）。

3．现在，单击**Create** 按钮（4）。此时会自动打开Scrapy数据库的仪表盘，在这里，你可以创建新的集合。

在Appery.io的术语中，一个数据库是由一组集合组成的。大致来说，一个应用使用一个单独的数据库（至少在最初时是这样），每个数据库中包含多个集合，比如用户、房产、消息等。Appery.io默认已经提供了一个**Users** 集合，其中包括用户名和密码（它们有很多内置功

能）。图4.2所示为创建集合的过程。

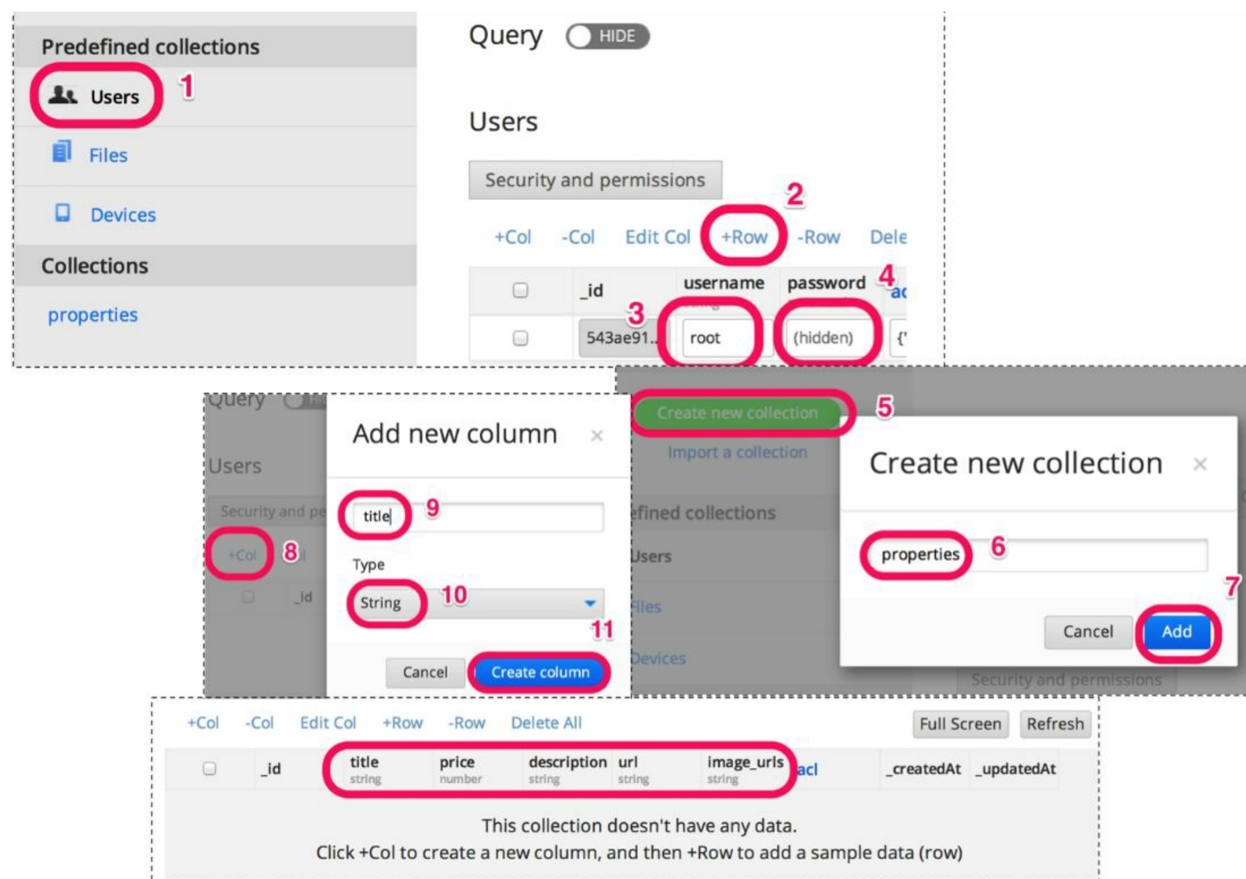


图4.2　使用Appery.io创建新数据库及集合

现在，我们添加一个用户，用户名为root，密码为pass。当然，你也可以选择更加安全的用户名和密码。为实现该目的，请单击侧边栏的**Users** 集合（1），然后单击**+Row** 添加用户/行（2）。在出现的两个字段中填入用户名和密码（3）和（4）。

我们还需要创建一个新的集合，用于存储Scrapy抓取到的房产数据，并将该集合命名为properties。通过单击绿色的**Create new collection** 按钮（5），将其命名为`properties`（6），然后单击**Add** 按钮

（7），就可以创建新的集合了。现在，我们还必须对该集合进行一些定制化处理。单击**+Col** 添加数据库列（8）。每个数据库列都有其类型，用于对值进行校验。除了价格是数值类型外，大部分字段都是简单的字符串类型。我们将通过单击**+Col** 添加几个列（8），并填充列名（9），如果不是字符串类型的话，还需要选择类型（10），然后单击**Create column** 按钮（11）。重复该过程5次，创建表4.1中展示的列。

表**4.1**

列	title	price	description	url	image_urls
类型	string	number	string	string	string

在集合创建的最后，你应该已经将所需的所有列都创建完成了，就像表4.1中所示的那样。现在已经准备好从Scrapy中导入一些数据了。

4.3　使用Scrapy填充数据库

首先，我们需要一个API key。我们可以在**Settings** 选项卡（1）中找到它。复制该值（2），然后单击**Collections** 选项卡（3）回到房产集合中，过程如图4.3所示。

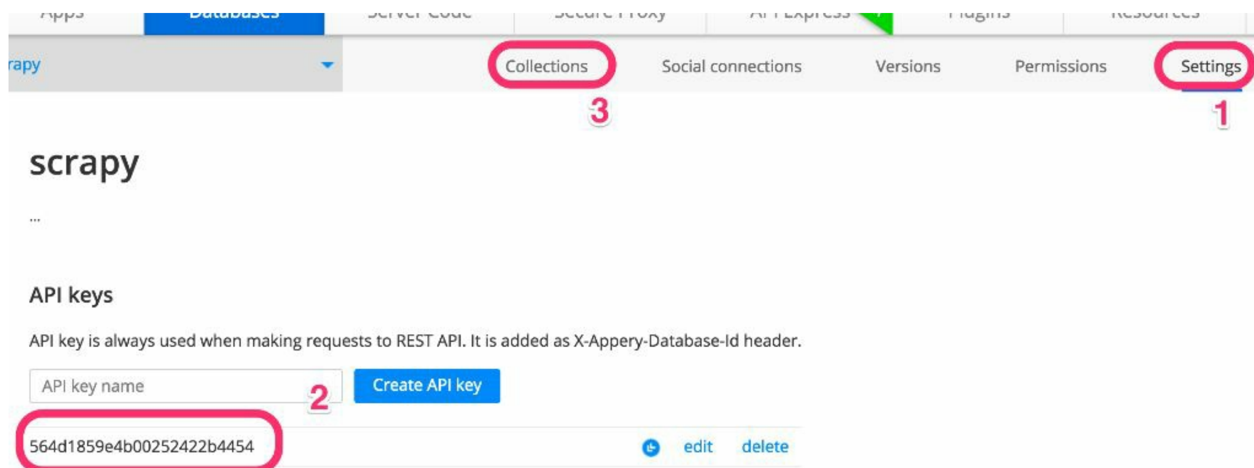


图4.3 使用Appery.io创建新数据库及集合

非常好！现在需要修改在第3章中创建的应用，将数据导入到Appery.io中。我们先将项目以及名为easy 的爬虫（easy.py ）复制过来，并将该爬虫重命名为tomobile （tomobile.py ）。同时，编辑文件，将其名称设为tomobile 。

```
$ ls

properties scrapy.cfg

$ cat properties/spiders/tomobile.py

...
```

```
class ToMobileSpider(CrawlSpider):

    name = 'tomobile'

    allowed_domains = ["scrapybook.s3.amazonaws.com"]

    # Start on the first index page

    start_urls = (

        'http://scrapybook.s3.amazonaws.com/properties/'

        'index_00000.html',

    )
```

```
...
```



本章代码可以在GitHub的`ch04` 目录下找到。

你可能已经注意到的一个问题是，这里并没有使用之前章节中用过的Web服务器（`http://web:9312` ），而是使用了该站点的一个公开可用的副本，这是我存放在`http://scrapybook.s3.amazonaws.com` 上的副本。之所以在本章中使用这种方式，是因为这样可以使图片和URL都能够公开可用，此时就可以非常轻松地分享应用了。

我们将使用Appery.io的管道来插入数据。Scrapy管道通常是一个很小的Python类，拥有后置处理、清理及存储Scrapy Item的功能。第8章将会更深入地介绍这部分的内容。就目前来说，你可以使用`easy_install` 或`pip` 安装它，不过如果你使用的是我们的Vagrant dev机器，则无需进行任何操作，因为我们已经将其安装好了。

```
$ sudo easy_install -U scrapyapperyio
```

或

```
$ sudo pip install --upgrade scrapyapperyio
```

此时，你需要对Scrapy的主设置文件进行一些小修改，将之前复制的API key添加进来。第7章将会更加深入地讨论设置。现在，我们所需要做的就是将如下行添加到`properties/settings.py` 文件中。

```
ITEM_PIPELINES = {'scrapyapperyio.ApperyIoPipeline': 300}

APPERYIO_DB_ID = '<<Your API KEY here>>'
APPERYIO_USERNAME = 'root'
APPERYIO_PASSWORD = 'pass'
APPERYIO_COLLECTION_NAME = 'properties'
```

不要忘记将APPERYIO_DB_ID替换为你的API key。此外，还需要确保设置中的用户名和密码，要和你在Appery.io中创建数据库用户时使用的相同。要想向Appery.io的数据库中填充数据，请像平常那样启动scrapy crawl。

```
$ scrapy crawl tomobile -s CLOSESPIDER_ITEMCOUNT=90

INFO: Scrapy 1.0.3 started (bot: properties)

...

INFO: Enabled item pipelines: ApperyIoPipeline

INFO: Spider opened

...

DEBUG: Crawled (200) <GET https://api.appery.io/rest/1/db/login?username=
```

```
root&password=pass>
...
DEBUG: Crawled (200) <POST https://api.appery.io/rest/1/db/collections/
properties>
...
INFO: Dumping Scrapy stats:
  {'downloader/response_count': 215,
   'item_scraped_count': 105,
```

```
  ...}

INFO: Spider closed (closespider_itemcount)
```

这次的输出会有些不同。可以看到在最开始的几行中，有一行是用于启用`ApperyIoPipeline`这个Item管道的；不过最明显的是，你会发现尽管抓取了100个Item，但是却有200次请求/响应。这是因为Appery.io的管道对每个Item都执行了一个到Appery.io服务端的额外请求，以便写入每一个Item。这些带有`api.appery.io`这个URL的请求同样也会在日志中出现。

当回到Appery.io时，可以看到在**properties**集合（1）中已经填充好了数据（2），如图4.4所示。

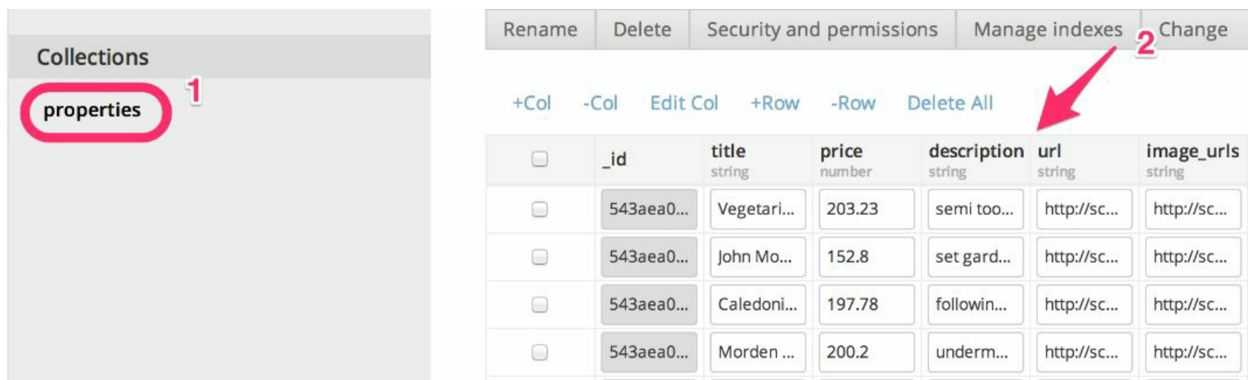


图4.4　使用数据填充properties集合

4.4　创建手机应用

创建一个新的手机应用非常简单。我们只需单击**Apps** 选项卡（1），然后单击绿色的**Create new app** 按钮（2）。填写应用名称为**properties** （3），然后单击**Create** 按钮进行创建就可以了，该过程如图4.5所示。

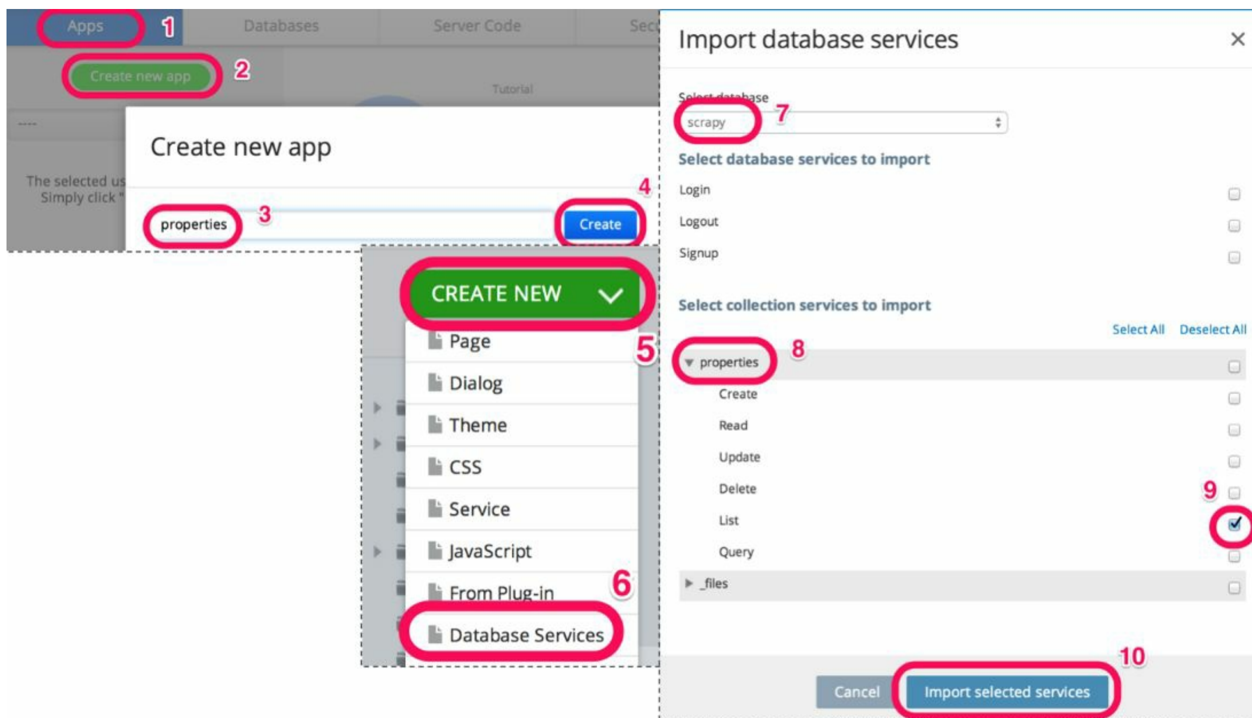


图4.5　创建新手机应用及数据库集合

4.4.1　创建数据库访问服务

创建新应用时的选项数量可能会有些多。使用Appery.io的应用编辑器，可以写出复杂的应用，不过我们将尽可能保持事情简单。我们最初需要的就是创建一个服务，能够让我们从应用中访问Scrapy数据库。为了达到这一目的，需要单击长方形的绿色按钮**CREATE NEW**（5），然后选择**Database Services**（6）。这时会弹出一个新的对话框，让我们选择想要连接的数据库。选择**scrapy** 数据库（7）。这个菜单中的大部分选项都不会用到，现在只需要单击展开**properties** 区域（8），然后选择**List**（9）。在后台，它会为我们编写代码，使得我们使用Scrapy爬取的数据可以在网络上使用。最后，单击**Import selected services** 按钮完成（10）。

4.4.2　创建用户界面

下面将要开始创建应用所有的可视化元素了，这将会使用编辑器中的**DESIGN** 选项卡来实现，如图4.6所示。

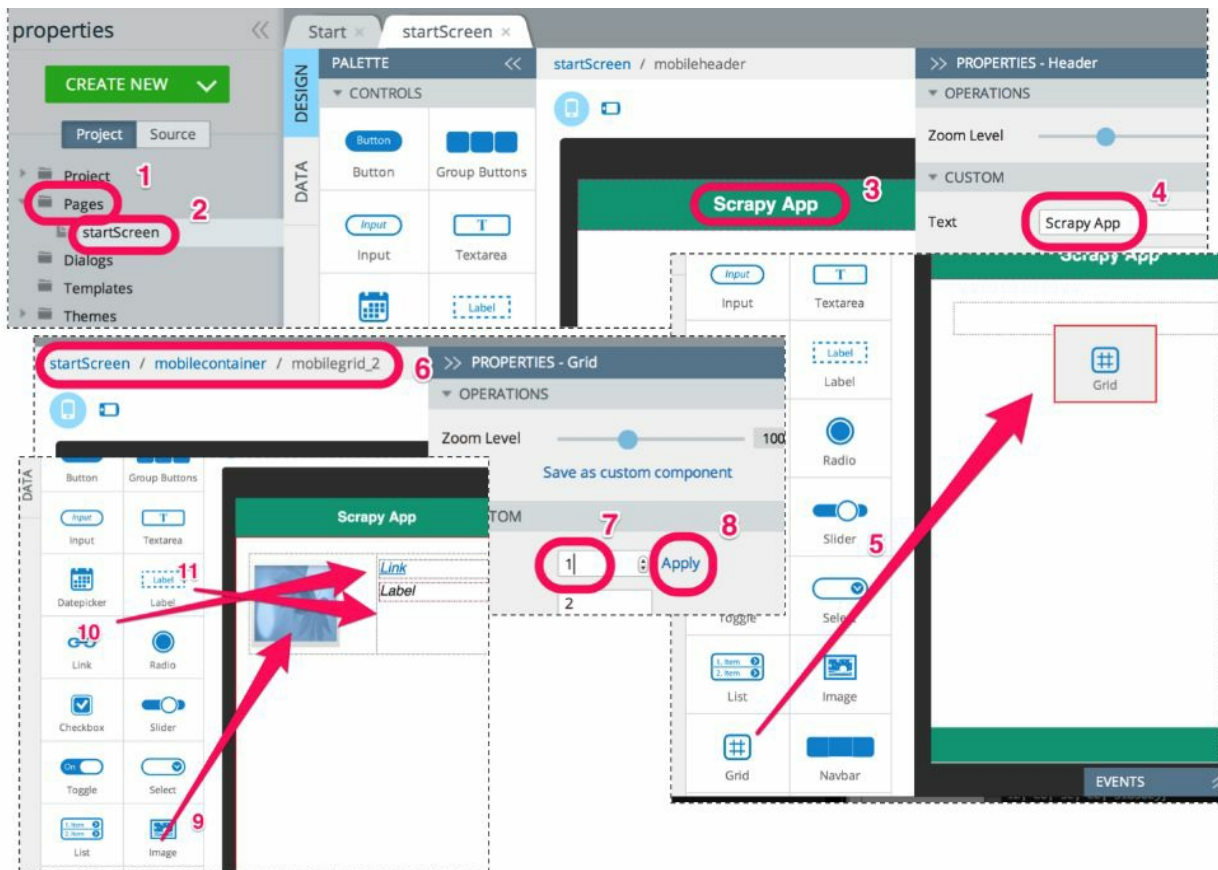


图4.6　创建用户界面

从页面左侧的树中，展开**Pages** 文件夹（1），然后单击**startScreen**（2）。UI编辑器将会打开该页面，我们可以在其中添加一些控件。下面使用编辑器编辑标题，以便对其更加熟悉。单击头部标题（3），然后会发现屏幕右侧的属性区域会变为显示标题的属性，其中包含一个`Text` 属性，将该属性值修改为**Scrapy App**，屏幕中间的标题也会相

应地更新。

然后，需要添加一个网格组件，从左侧面板（5）中拖曳**Grid** 控件即可实现。该控件有两行，而根据我们的需求，只需要一行即可。选择刚刚添加的网格。当手机视图顶部的缩略图区域（6）变灰时，就可以知道该网格已经被选取了。如果没有被选取，单击该网格以便选中。然后右侧的属性栏会更新为网格的属性。这里只需要将Rows属性设置为1，然后单击Apply即可（7）和（8）。现在，该网格就会被更新为只有一行了。

最后，拖拽另外一些控件到网格中。首先要在网格左侧添加图片控件（9），然后在网格右侧添加链接（10），最后在链接下面添加标签（11）。

就布局而言，此时已经足够。接下来将从数据库中向用户界面输入数据。

4.4.3　将数据映射到用户界面

目前为止，我们花费了大量时间在DESIGN选项卡中，以创建应用的可视化效果。为了将可用的数据链接到这些控件中，需要切换到**DATA** 选项卡（1），如图4.7所示。

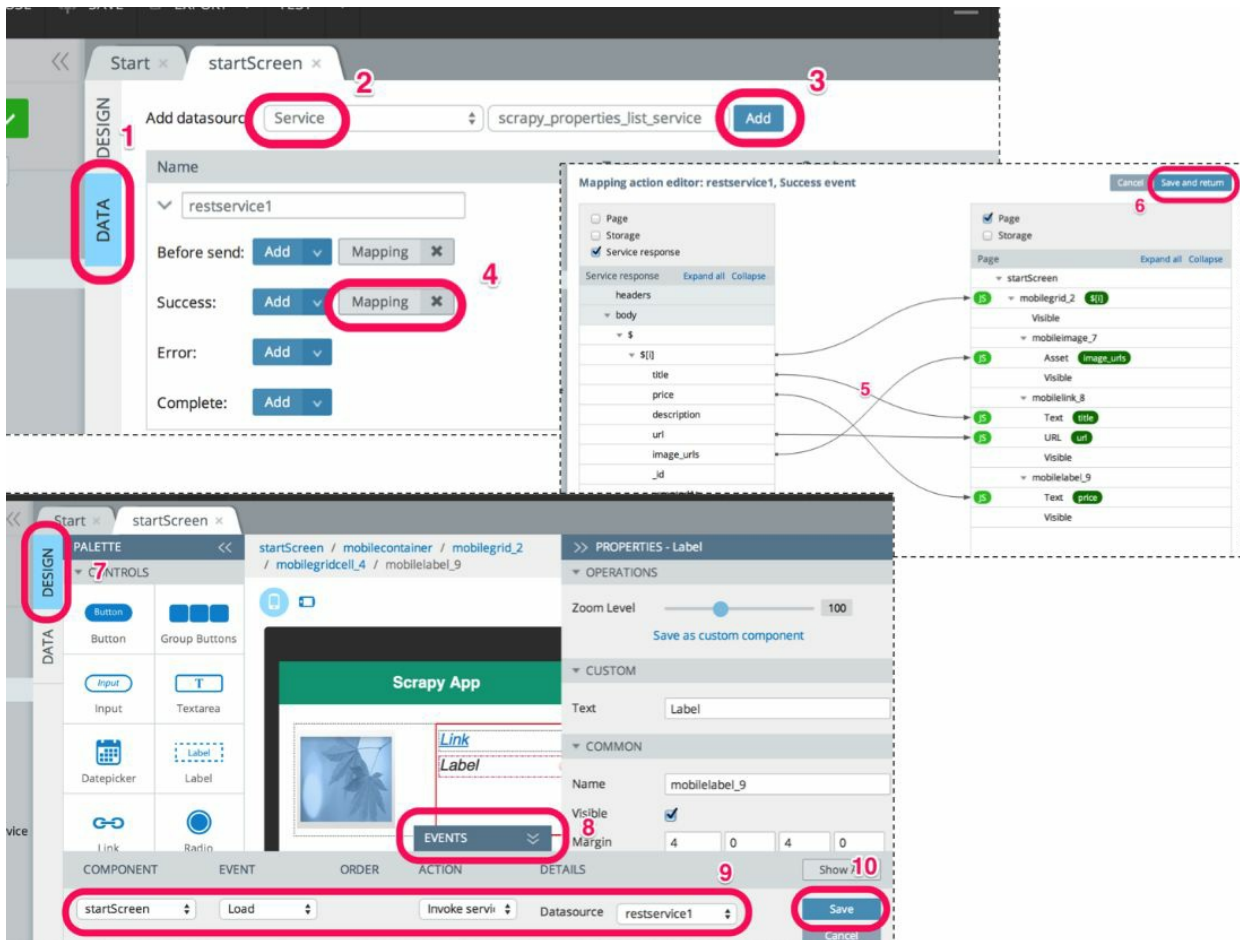


图4.7　将数据映射到用户界面

选择**Service**（2）作为数据源类型。由于前面创建的服务是唯一可用的服务，因此它会被自动选取。然后可以继续单击**Add** 按钮（3），此时服务属性将会在其下方列出。只要按下了**Add** 按钮，就会看到像**Before send** 以及**Success** 这样的事件。我们可以通过单击**Success** 后面的**Mapping** 按钮，定制服务成功调用后要做的事情。

此时会打开**Mapping action editor**，我们可以在这里完成连线。该编辑器有两侧。左侧是服务响应中可用的字段，而在右侧中可以看到前面步骤中添加的UI控件的属性。两侧都有一个**Expand all** 链接，单击该链接可以看到所有可用的数据和控件。接下来，需要按照表4.2中给出

的5个映射，从左侧向右侧拖曳。

表**4.2**

响应	组件	属性	备注
$[i]	mobilegrid_2		使用for循环创建每一行
title	mobilelink_8	Text	设置链接文本
price	mobilelabel_9	Text	在文本域中设置价格
image_urls	mobileimage_7	Asset	从图片容器的URL中加载图片
url	mobilelink_8	URL	为链接设置URL。当用户单击时，将会加载关联的页面

4.4.4　数据库字段与用户界面控件间映射

表4.2中项的数量可能会与你的情况有些许差别，不过由于每种控件都只有一个，因此出错的可能性非常小。通过设置这些映射，我们通知Appery.io在后台编写所有代码，以便在数据库查询成功时使用数据库中的值加载控件。下面，可以单击**Save and return** 按钮（6）继续。

此时又回到了**DATA** 选项卡，如图4.7所示。由于还需要返回到UI编辑器当中，因此需要单击**DESIGN** 选项卡（7）。在屏幕下方，你会发现一个**EVENTS** 区域（8），尽管该区域一直存在，但它刚刚才被展

开。在**EVENTS** 区域中，我们让Appery.io做一些事情，作为对UI事件的响应。这是我们需要执行的最后一个步骤。它会让应用在UI加载完成后立即调用服务取回数据。为了实现该功能，我们需要选择**startScreen**作为组件，并将事件保持为默认的**Load** 选项。然后选择**Invoke service**作为**action**， 保持**Datasource** 为默认的**restservice1** 选项（9）。最后，单击**Save** （10），这就是我们为创建这个手机应用所做的所有事情了。

4.4.5　测试、分享及导出你的手机应用

现在，可以测试这个应用了。我们所需要做的事情就是单击UI生成器顶部的**TEST** 按钮（1），如图4.8所示。

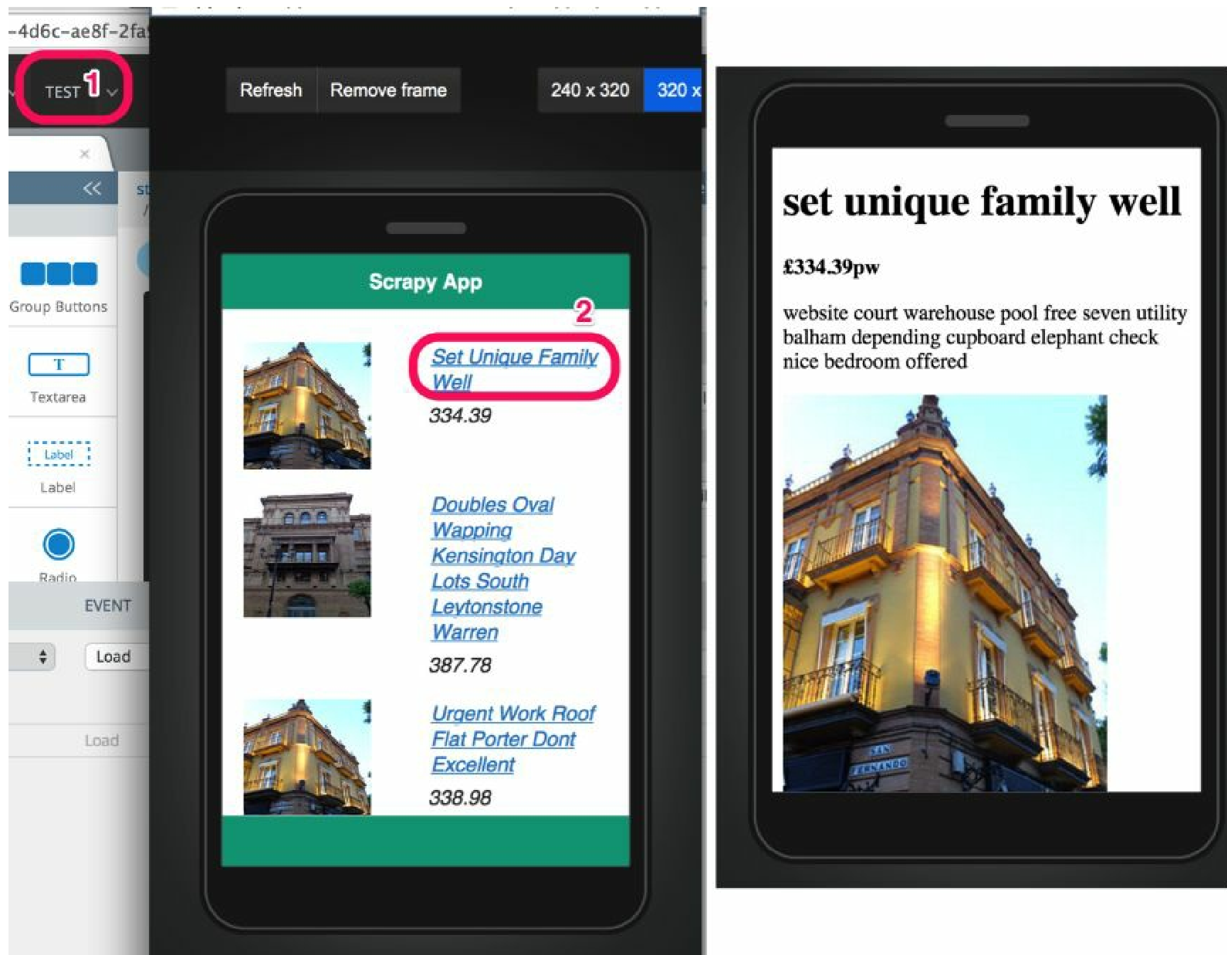


图4.8　运行在你浏览器中的手机应用

手机应用将会在浏览器中运行。这些链接都是有效的（2），可以浏览。可以预览不同的手机屏幕方案以及设备方向，也可以单击**View on Phone** 按钮，此时会显示一个二维码，你可以使用移动设备扫描该二维码，并预览该应用。你只需分享其生成的链接，其他人也可以在他们的浏览器中尝试该应用。

只需单击几下，我们就可以将Scrapy抓取的数据组织起来，并展示在手机应用中。如果你需要更进一步地定制该应用，可以参考Appery.io提供的教程，其网址为`http://devcenter.appery.io/tutorials/`

。当一切准备就绪时，就可以通过**EXPORT** 按钮导出该应用了，Appery.io提供了非常丰富的导出选项，如图4.9所示。

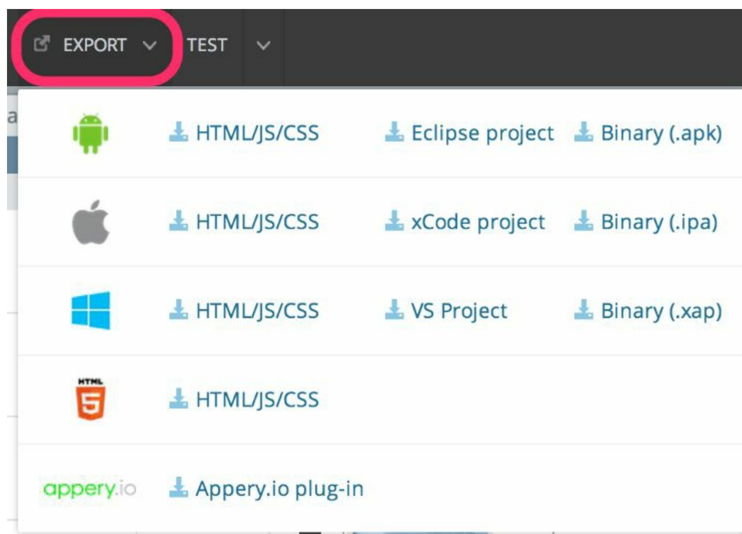


图4.9　你可以将应用导出到大部分主流移动平台

你可以导出项目文件，在自己喜欢的IDE中进一步开发；也可以获得二进制文件，发布到各个平台的手机市场当中。

4.5　本章小结

使用Scrapy和Appery.io这两个工具，我们拥有了一个可以抓取网站并且能够将数据插入到数据库中的系统。此外，我们还得到了RESTful API，以及一个简单的可以用于Android和iOS的手机应用。对于高级特性和进一步开发，你可以更加深入到这些平台中，将其中部分开发工作外包给领域专家，或是研究替代方案。现在，你只需要最少的编码，就能够拥有一个可以演示应用理念的最小产品。

你会注意到，在如此短的开发时间中，我们的应用看起来还不错。

这是因为它使用了真实的数据，而不是占位符，并且所有链接都是可用且有意义的。我们成功创建了一个尊重其生态（源网站）的最小可用产品，并以流量的形式将价值回馈给源网站。

现在，我们可以开始学习如何使用Scrapy爬虫在更加复杂的场景下抽取数据了。

第5章　迅速的爬虫技巧

第3章关注的是如何从页面中抽取信息，并将其存储到`Items`中。我们所学习的内容已经覆盖了大部分常见的Scrapy用例，足够你创建并运行爬虫了。而在本章中，我们将看到更多特殊的例子，以便让你更加熟悉Scrapy的两个最重要的类——`Request`和`Response`，即我们在第3章中提到的UR^{2}IM抓取模型中的两个R。

5.1　需要登录的爬虫

通常情况下，你会发现自己想要抽取数据的网站存在登录机制。大部分情况下，网站会要求你提供用户名和密码用于登录。你可以从`http://web:9312/dynamic`（从dev机器访问）或`http://localhost:9312/ dynamic`（从宿主机浏览器访问）找到我们要使用的例子。如果使用"user"作为用户名，"pass"作为密码的话，你就可以访问到包含3个房产页面链接的网页。不过现在的问题是，要如何使用Scrapy执行相同的操作？

让我们使用Google Chrome浏览器的开发者工具来尝试理解登录的工作过程（见图5.1）。首先，打开**Network**选项卡（1）。然后，填写用户名和密码，并单击**Login**（2）。如果用户名和密码正确，你将会看到包含3个链接的页面。如果用户名和密码不匹配，将会看到一个错误页。

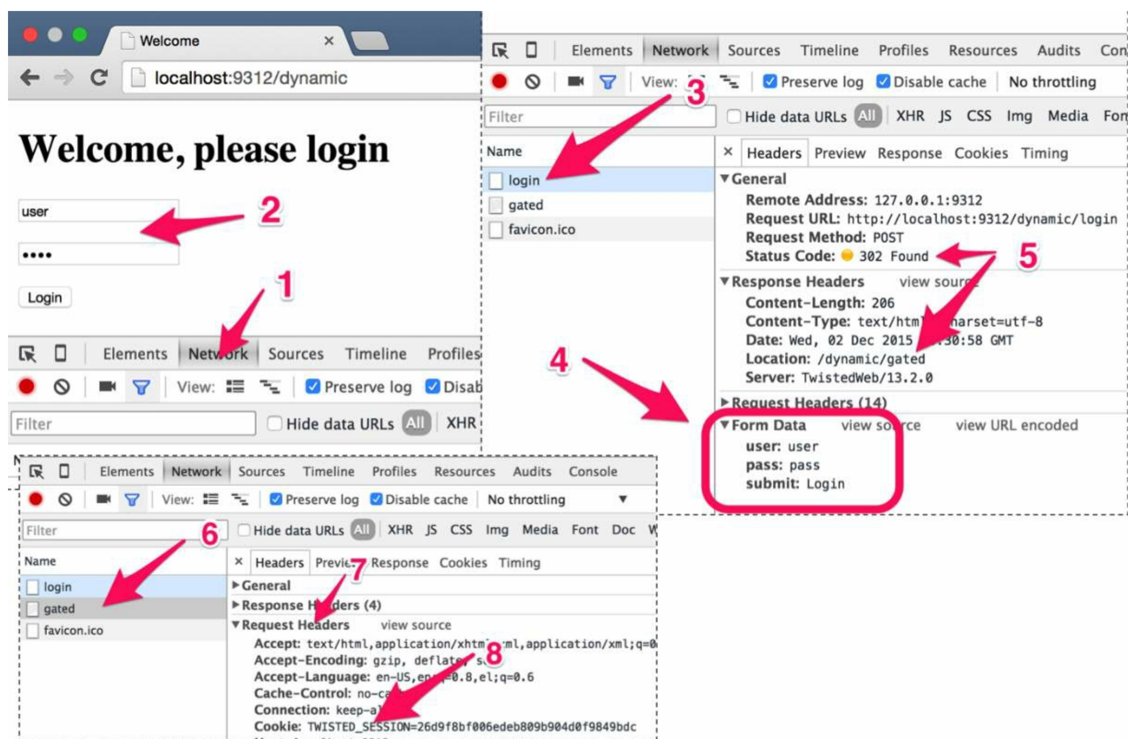


图5.1　登录网站时的请求和响应

当按下**Login**按钮时，会在Google Chrome浏览器开发者工具的**Network**选项卡中看到一个包含**Request Method: POST**的请求，其目的地址为`http://localhost:9312/dynamic/login`。



> 前面章节中的请求都是GET类型的请求，一般用于获取不会改变的数据，比如简单的网页、图像等。而POST类型的请求通常用于获取那些依赖于传送给服务器内容的数据，比如本例中的用户名和密码。

当你单击该请求时（3），可以看到发送给服务端的数据，包括**Form Data**（4），其中包含了我们输入的用户名和密码。这些数据都

是以文本形式传输给服务端的。Chrome浏览器只是将其组织起来，向我们更好地显示这些数据。服务端的响应是**302 Found** （5），使我们跳转到一个新的页面：`/dynamic/gated` 。该页面只有在登录成功后才会出现。如果尝试直接访问`http://localhost:9312/dynamic/gated`，而不输入正确的用户名和密码的话，服务端会发现你在作弊，并跳转到错误页，其地址是`http:// localhost:9312/dynamic/error` 。服务端是如何知道你和你的密码的呢？如果你单击开发者工具左侧的**gated** （6），就会发现在**Request Headers** 区域下面（7）设置了一个**Cookie** 值（8）。



HTTP Cookie是一些服务端发送给浏览器的文本或数值，通常都很短。相应地，浏览器会在随后的每个请求中将其返回给服务端，用于标识你、用户和会话。这样你就可以执行需要服务端状态信息的复杂操作了，比如购物车里的商品或你的用户名和密码。

总之，即使是一个单一的操作，比如登录，也可能涉及包括POST请求和HTTP跳转的多次服务端往返。Scrapy能够自动处理大部分操作，而我们需要编写的代码也很简单。

我们从第3章中名为`easy` 的爬虫开始，创建一个新的爬虫，命名为`login` ，保留原有文件，并修改爬虫中的`name` 属性（如下所示）：

```
class LoginSpider(CrawlSpider):
    name = 'login'
```



本章代码在GitHub的`ch05` 目录下，其中本示例为`ch05/properties` 。

我们需要通过执行到`http://localhost:9312/dynamic/login` 的POST请求，发送登录的初始请求。这将通过Scrapy的`FormRequest` 类实现该功能。该类与第3章中使用的`Request` 类相似，不过该类额外包含一个`formdata` 参数，可以使用该参数传输表单数据（`user` 和`pass` ）。要想使用该类，首先需要引入如下模块。

```
from scrapy.http import FormRequest
```

然后，将`start_urls` 语句替换为`start_requests()` 方法。这样做是因为在本例中，我们需要从一些更加定制化的请求开始，而不仅仅是几个URL。更确切地说就是，我们从该函数中创建并返回一个`FormRequest` 。

```
# Start with a login request
def start_requests(self):
  return [
    FormRequest(
      "http://web:9312/dynamic/login",
      formdata={"user": "user", "pass": "pass"}
         )]
```

虽然听起来不可思议，但是`CrawlSpider`（`LoginSpider`的基类）默认的`parse()`方法确实处理了`Response`，并且仍然能够使用第3章中的`Rule`和`LinkExtractor`。我们只编写了非常少的额外代码，这是因为Scrapy为我们透明处理了Cookie，并且一旦我们登录成功，就会在后续的请求中传输这些Cookie，就和浏览器执行的方式一样。接下来可以像平常一样，使用`scrapy crwal`运行。

```
$ scrapy crawl login

INFO: Scrapy 1.0.3 started (bot: properties)

...

DEBUG: Redirecting (302) to <GET .../gated> from <POST .../login >

DEBUG: Crawled (200) <GET .../data.php>
```

```
DEBUG: Crawled (200) <GET .../property_000001.html> (referer: .../data.
php)
DEBUG: Scraped from <200 .../property_000001.html>
{'address': [u'Plaistow, London'],
 'date': [datetime.datetime(2015, 11, 25, 12, 7, 27, 120119)],
 'description': [u'features'],
 'image_urls': [u'http://web:9312/images/i02.jpg'],
...
```

```
INFO: Closing spider (finished)

INFO: Dumping Scrapy stats:

  {...

   'downloader/request_method_count/GET': 4,

   'downloader/request_method_count/POST': 1,

...

   'item_scraped_count': 3,
```

我们可以在日志中看到从`dynamic/login`到`dynamic/gated`的跳转，然后就会像平时那样抓取Item了。在统计中，可以看到1个POST请求和4个GET请求（一个是前往`dynamic/gated`索引页，另外3个是房产页面）。



> 本例中，我们没有保护房产页面本身，而是只保护了到这些页面的链接。无论哪种情况，前面的代码都是适用的。

如果使用了错误的用户名和密码，将会跳转到一个没有任何项目的页面，并且此时爬取过程会被终止，如下面的执行情况所示。

```
$ scrapy crawl login

INFO: Scrapy 1.0.3 started (bot: properties)

...
```

```
DEBUG: Redirecting (302) to <GET .../dynamic/error > from <POST .../
dynamic/login>
DEBUG: Crawled (200) <GET .../dynamic/error>
...
INFO: Spider closed (closespider_itemcount)
```

这是一个简单的登录示例，用于演示基本的登录机制。大多数网站都会拥有一些更加复杂的机制，不过Scrapy也都能够轻松处理。比如，一些网站要求你在执行POST请求时，将表单页中的某些表单变量传输到登录页，以便确认Cookie是启用的，同样也会让你在尝试暴力破解成千上万次用户名/密码的组合时更加困难。图5.2所示即为此种情况的一个示例。

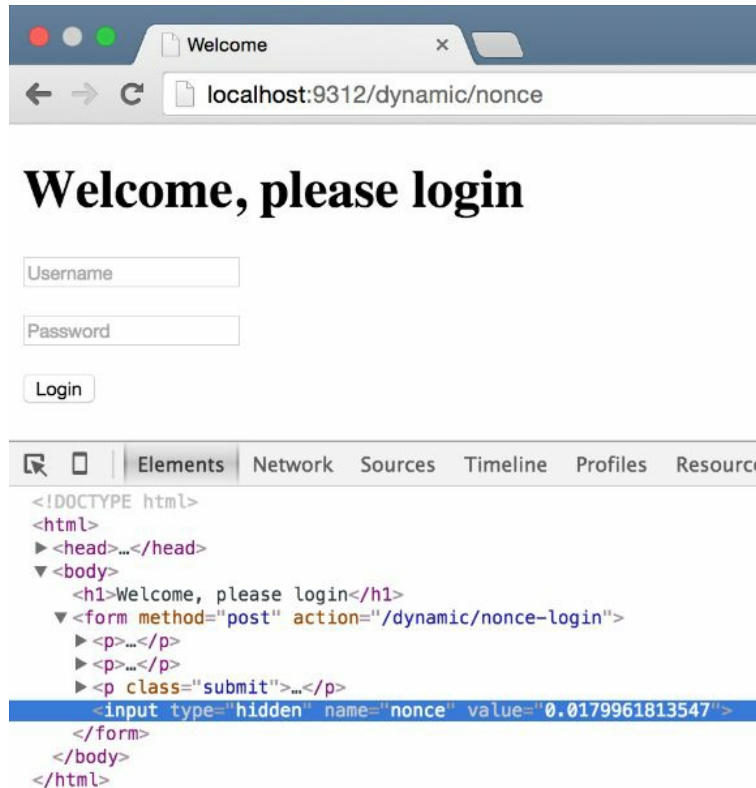


图5.2　使用一次性随机数的一个更加高级的登录示例的请求和响应情况

比如，当访问`http://localhost:9312/dynamic/nonce`时，你会看到一个看起来一样的页面，但是当使用Chrome浏览器的开发者工具查看时，会发现页面的表单中有一个叫作**nonce**的隐藏字段。当提交该表单时（提交到`http://localhost:9312/ dynamic/nonce-login`），除非你既传输了正确的用户名/密码，又提交了服务端在你访问该登录页时给你的`nonce`值，否则登录不会成功。你无法猜测该值，因为它通常是随机且一次性的。这就表示要想成功登录，现在就需要请求两次了。你必须先访问表单页，然后再访问登录页传输数据。当然，Scrapy同样拥有内置函数可以帮助我们实现这一目的。

我们创建了一个和之前相似的`NonceLoginSpider`爬虫。现在，在`start_requests()`中，将返回一个简单的`Request`（不要忘记引入

该模块）到表单页面中，并通过设置其callback 属性为处理方法parse_welcome() 手动处理响应。在parse_welcome() 中，使用了FormRequest 对象的辅助方法from_response()，以创建从原始表单中预填充所有字段和值的FormRequest 对象。FormRequest.from_response() 粗略模拟了一次在页面的第一个表单上的提交单击，此时所有字段留空。



花费一些时间让自己熟悉from_response() 的文档是值得的。它有很多非常有用的功能，如formname 和formnumber 可以帮助你在拥有多个表单的页面上选择其中某个表单。

该方法对于我们来说非常有用，因为它能够毫不费力地原样包含表单中的所有隐藏字段。我们所需要做的就是使用formdata 参数填充user 和pass 字段以及返回FormRequest 。下面是其相关代码。

```
# Start on the welcome page
def start_requests(self):
    return [
        Request(
            "http://web:9312/dynamic/nonce",
            callback=self.parse_welcome)
    ]

# Post welcome page's first form with the given user/pass
def parse_welcome(self, response):
    return FormRequest.from_response(
        response,
        formdata={"user": "user", "pass": "pass"}
    )
```

我们可以像平时一样运行爬虫。

```
$ scrapy crawl noncelogin

INFO: Scrapy 1.0.3 started (bot: properties)

...

DEBUG: Crawled (200) <GET .../dynamic/nonce>

DEBUG: Redirecting (302) to <GET .../dynamic/gated > from <POST .../

dynamic/login-nonce>

DEBUG: Crawled (200) <GET .../dynamic/gated>
```

```
...
INFO: Dumping Scrapy stats:
  {...
   'downloader/request_method_count/GET': 5,
   'downloader/request_method_count/POST': 1,
...
   'item_scraped_count': 3,
```

可以看到，第一个GET请求前往`/dynamic/nonce`页面，然后是POST请求，跳转到`/dynamic/nonce-login`页面，之后像前面的例子一样跳转到`/dynamic/gated`页面。关于登录的讨论就到这里。该示例使用两个步骤完成登录。只要你有足够的耐心，就可以形成任意长链，来执行几乎所有的登录操作。

5.2　使用JSON API和AJAX页面的爬虫

有时，你会发现自己在页面寻找的数据无法从HTML页面中找到。比如，当访问`http://localhost:9312/static/`时（见图5.3），在页面任意位置右键单击**inspect element**（1, 2），可以看到其中包含所有常见HTML元素的DOM树。但是，当你使用`scrapy shell`请求，或是在Chrome浏览器中右键单击**View Page Source**（3, 4）时，则会发现该页面的HTML代码中并不包含关于房产的任何信息。那么，这些数据是从哪里来的呢？

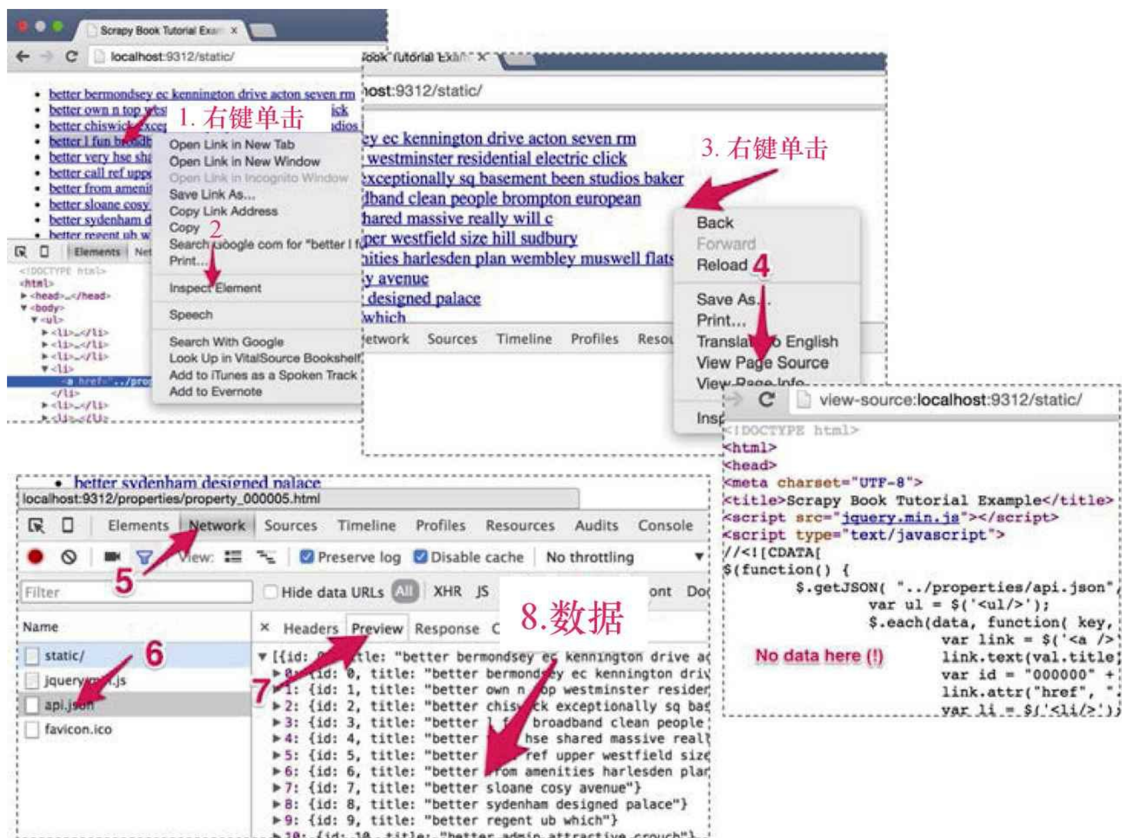


图5.3　动态加载JSON对象时的页面请求与响应

与平常一样，遇到这类例子时，下一步操作应当是打开Chrome浏览器开发者工具的**Network** 选项卡，来看看发生了什么。在左侧的列表中，可以看到加载本页面时Chrome执行的请求。在这个简单的页面中，只有3个请求：**static/** 是刚才已经检查过的请求；**jquery.min.js** 用于获取一个流行的Javascript框架的代码；而**api.json** 看起来会让我们产生兴趣。当单击该请求（6），并单击右侧的**Preview** 选项卡（7）时，就会发现这里面包含了我们正在寻找的数据。实际上，`http://localhost:9312/properties/api.json` 包含了房产的ID和名称（8），如下所示。

```
[{
    "id": 0,
```

```
    "title": "better set unique family well"
},
... {
    "id": 29,
    "title": "better portered mile"
}]
```

这是一个非常简单的JSON API的示例。更复杂的API可能需要你登录，使用POST请求，或返回更有趣的数据结构。无论在哪种情况下，JSON都是最简单的解析格式之一，因为你不需要编写任何XPath表达式就可以从中抽取出数据。

Python提供了一个非常好的JSON解析库。当我们执行`import json`时，就可以使用`json.loads(response.body)`解析JSON，将其转换为由Python原语、列表和字典组成的等效Python对象。

我们将第3章的`manual.py`拷贝过来，用于实现该功能。在本例中，这是最佳的起始选项，因为我们需要通过在JSON对象中找到的ID，手动创建房产URL以及`Request`对象。我们将该文件重命名为`api.py`，并将爬虫类重命名为`ApiSpider`，`name`属性修改为`api`。新的`start_urls`将会是JSON API的URL，如下所示。

```
start_urls = (
    'http://web:9312/properties/api.json',
)
```

如果你想执行POST请求，或是更复杂的操作，可以使用前一节中介绍的`start_requests()`方法。此时，Scrapy将会打开该URL，并调用包含以`Response`为参数的`parse()`方法。可以通过`import json`，使用如下代码解析JSON对象。

```
def parse(self, response):
    base_url = "http://web:9312/properties/"
    js = json.loads(response.body)
    for item in js:
        id = item["id"]
        url = base_url + "property_%06d.html" % id
        yield Request(url, callback=self.parse_item)
```

前面的代码使用了`json.loads(response.body)`，将`Response`这个JSON对象解析为Python列表，然后迭代该列表。对于列表中的每一项，我们将URL的3个部分（`base_url`、`property_%06d`以及`.html`）组合到一起。`base_url`是在前面定义的URL前缀。`%06d`是Python语法中非常有用的一部分，它可以让我们结合Python变量创建新的字符串。在本例中，`%06d`将会被变量`id`的值替换（本行结尾处%后面的变量）。`id`将会被视为数字（`%d`表示视为数字），并且如果不满6位，则会在前面加上0，扩展成6位字符。比如，`id`值为5，`%06d`将会被替换为000005，而如果`id`为34322，`%06d`则会被替换为034322。最终结果正是我们房产页面的有效URL。我们使用该URL形成一个新的`Request`对象，并像第3章一样使用`yield`。然后可以像平时那样使用`scrapy crawl`运行该示例。

```
$ scrapy crawl api

INFO: Scrapy 1.0.3 started (bot: properties)

...

DEBUG: Crawled (200) <GET ...properties/api.json>

DEBUG: Crawled (200) <GET .../property_000029.html>

...

INFO: Closing spider (finished)

INFO: Dumping Scrapy stats:
```

```
...

  'downloader/request_count': 31, ...

  'item_scraped_count': 30,
```

你可能会注意到结尾处的状态是31个请求——每个Item一个请求，以及最初的api.json 的请求。

5.2.1 在响应间传参

很多情况下，在JSON API中会有感兴趣的信息，你可能想要将它们存储到Item 中。在我们的示例中，为了演示这种情况，JSON API会在给定房产信息的标题前面加上"better"。比如，房产标题是"Covent Garden"，API就会将标题写为"Better Covent Garden"。假设我们想要将这些"better"开头的标题存储到Items 中，要如何将信息从parse() 方法传递到parse_item() 方法呢？

不要感到惊讶，通过在parse() 生成的Request 中设置一些东西，

就能实现该功能。之后，可以从parse_item()接收到的Response中取得这些信息。Request有一个名为meta的字典，能够直接访问Response。比如在我们的例子中，可以在该字典中设置标题值，以存储来自JSON对象的标题。

```
title = item["title"]
yield Request(url, meta={"title": title},callback=self.parse_item)
```

在parse_item()内部，可以使用该值替代之前使用过的XPath表达式。

```
l.add_value('title', response.meta['title'],
            MapCompose(unicode.strip, unicode.title))
```

你会发现我们不再调用add_xpath()，而是转为调用add_value()，这是因为我们在该字段中将不会再使用到任何XPath表达式。现在，可以使用scrapy crawl运行这个新的爬虫，并且可以在PropertyItems中看到来自api.json的标题。

5.3　30倍速的房产爬虫

有这样一种趋势，当你开始使用一个框架时，做任何事情都可能会使用最复杂的方式。你在使用Scrapy时也会发现自己在做这样的事情。

在疯狂于XPath等技术之前，值得停下来想一想：我选择的方式是从网站中抽取数据最简单的方式吗？

如果你能从索引页中抽取出基本相同的信息，就可以避免抓取每个房源页，从而得到数量级的提升。



请记住，很多网站在其索引页中提供了不同的项目数量选择。比如，一个网站可能允许你通过调整参数指定每个索引页显示的房源数是10、50还是100，如`&show=50`。显然，如果是这样的情况，就可以将该参数设置为允许的最大值。

比如，在房产示例中，我们所需要的所有信息都存在于索引页中，包括标题、描述、价格和图片。这就意味着只抓取一个索引页，就能抽取其中的30个条目以及前往下一页的链接。通过爬取100个索引页，我们只需要100个请求，而不是3000个请求，就能够得到3000个条目。太棒了！

在真实的Gumtree网站中，索引页的描述信息要比列表页中完整的描述信息稍短一些。不过此时这种抓取方式可能也是可行的，甚至也能令人满意。



在许多情况下，我们将不得不权衡数据质量与请求数量的关系。很多源

都会限制大量的请求（后续章节会遇到更多此类问题），因此在索引中获取也可能帮助我们解决其他难题。

在我们的例子中，当查看任何一个索引页的HTML代码时，就会发现索引页中的每个房源都有其自己的节点，并使用`itemtype="http://schema.org/Product"`来表示。在该节点中，我们拥有与详情页完全相同的方式为每个属性注解的所有信息，如图5.4所示。

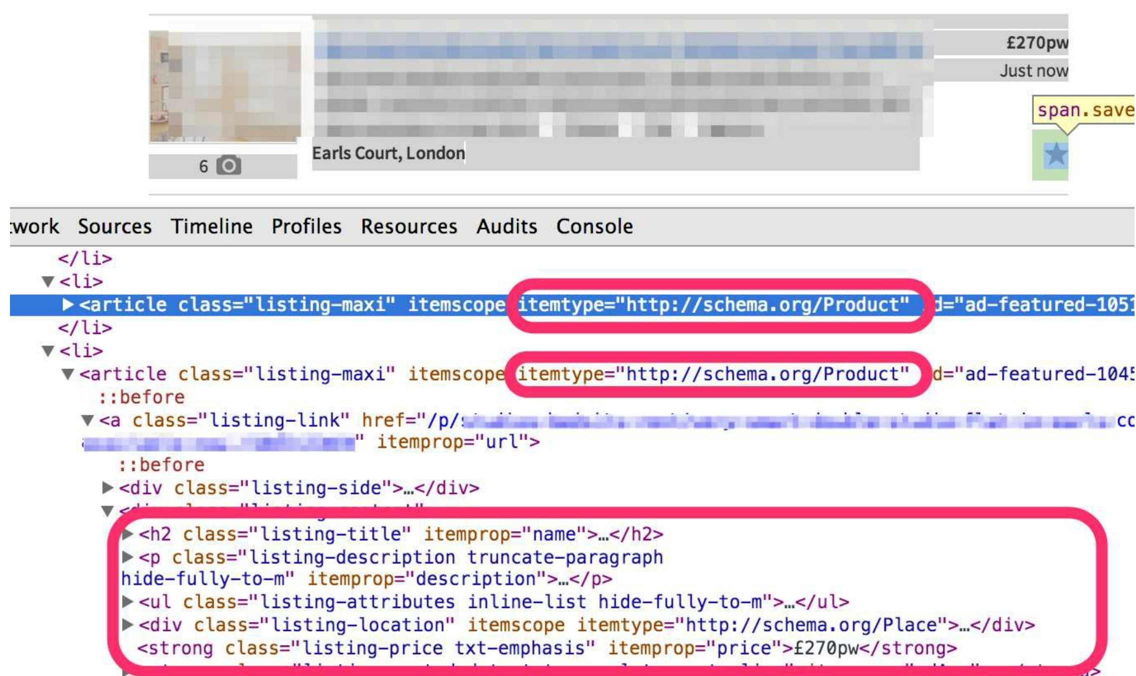


图5.4　从单一索引页抽取多个房产信息

我们在Scrapy shell中加载第一个索引页，并使用XPath表达式进行测试。

```
$ scrapy shell http://web:9312/properties/index_00000.html
```

在Scrapy shell中，尝试选取所有带有Product标签的内容：

```
>>> p=response.xpath('//*[@itemtype="http://schema.org/Product"]')

>>> len(p)

30

>>> p

[<Selector xpath='//*[@itemtype="http://schema.org/Product"]' data=u'<li

class="listing-maxi" itemscopeitemt'...]
```

可以看到我们得到了一个包含30个`Selector`对象的列表，每个对象指向一个房源。在某种意义上，`Selector`对象与`Response`对象有些相似，我们可以在其中使用XPath表达式，并且只从它们指向的地方获取信息。唯一需要说明的是，这些表达式应该是相对XPath表达式。相对XPath表达式与我们之前看到的基本一样，不过在前面增加了一个'.'点号。举例说明，让我们看一下使用`.//*[@itemprop="name"][1]/text()`这个相对XPath表达式，从第4个房源抽取标题时是如何工作的。

```
>>> selector = p[3]

>>> selector

<Selector xpath='//*[@itemtype="http://schema.org/Product"]' ... '>

>>> selector.xpath('.//*[@itemprop="name"][1]/text()').extract()
```

```
[u'l fun broadband clean people brompton european']
```

可以在Selector 对象的列表中使用for 循环，抽取索引页中全部30个条目的信息。

为了实现该目的，我们再一次从第3章的manual.py 着手，将爬虫重命名为"fast"，并重命名文件为fast.py 。我们将复用大部分代码，只在parse() 和parse_items() 方法中进行少量修改。最新方法的代码如下。

```
def parse(self, response):
    # Get the next index URLs and yield Requests
    next_sel = response.xpath('//*[contains(@class,"next")]//@href')
    for url in next_sel.extract():
        yield Request(urlparse.urljoin(response.url, url))

    # Iterate through products and create PropertiesItems
    selectors = response.xpath(
        '//*[@itemtype="http://schema.org/Product"]')
    for selector in selectors:
        yield self.parse_item(selector, response)
```

在代码的第一部分中，对前往下一个索引页的Request 的yield 操

作的代码没有变化。唯一改变的内容在第二部分，不再使用yield为每个详情页创建请求，而是迭代选择器并调用parse_item()。其中，parse_item()的代码也和原始代码非常相似，如下所示。

```
def parse_item(self, selector, response):
    # Create the loader using the selector
    l = ItemLoader(item=PropertiesItem(), selector=selector)

    # Load fields using XPath expressions
    l.add_xpath('title', './/*[@itemprop="name"][1]/text()',
                MapCompose(unicode.strip, unicode.title))
    l.add_xpath('price', './/*[@itemprop="price"][1]/text()',
                MapCompose(lambda i: i.replace(',', ''), float),
                re='[,.0-9]+')
    l.add_xpath('description',
                './/*[@itemprop="description"][1]/text()',
                MapCompose(unicode.strip), Join())
    l.add_xpath('address',
                './/*[@itemtype="http://schema.org/Place"]'
                '[1]/*/text()',
                MapCompose(unicode.strip))
    make_url = lambda i: urlparse.urljoin(response.url, i)
    l.add_xpath('image_urls', './/*[@itemprop="image"][1]/@src',
                MapCompose(make_url))

    # Housekeeping fields
    l.add_xpath('url', './/*[@itemprop="url"][1]/@href',
                MapCompose(make_url))
    l.add_value('project', self.settings.get('BOT_NAME'))
    l.add_value('spider', self.name)
    l.add_value('server', socket.gethostname())
    l.add_value('date', datetime.datetime.now())

    return l.load_item()
```

我们所做的细微变更如下所示。

- ItemLoader现在使用selector作为源，而不再是Response。这

是`ItemLoader` API一个非常便捷的功能，能够让我们从当前选取的部分（而不是整个页面）抽取数据。

- XPath表达式通过使用前缀点号（.）转为相对XPath。



比较巧合的是，在我们的例子中，索引页和详情页中的XPath表达式是一样的。实际情况并不总是这样，你可能需要重新开发XPath表达式，以匹配索引页的结构。

- 我们必须自己编辑`Item`的URL。之前，`response.url`已经给出了房源页的URL。而现在，它给出的是索引页的URL，因为该页面才是我们要爬取的。我们需要使用熟悉的`.//*[@itemprop="url"][1]/@href`这个XPath表达式抽取出房源的URL，然后使用`MapCompose`处理器将其转换为绝对URL。

小的改变能够节省巨大的工作量。现在，我们可以使用如下代码运行该爬虫。

```
$ scrapy crawl fast -s CLOSESPIDER_PAGECOUNT=3

...
```

```
INFO: Dumping Scrapy stats:

   'downloader/request_count': 3, ...

   'item_scraped_count': 90,...
```

和预期一样，只用了3个请求，就抓取了90个条目。如果我们没有在索引页中获取到的话，则需要93个请求。这种方式太明智了！

如果你想使用scrapy parse 进行调试，那么现在必须设置spider参数，如下所示。

```
$ scrapy parse --spider=fast http://web:9312/properties/index_00000.html

...
```

```
>>> STATUS DEPTH LEVEL 1 <<<

# Scraped Items --------------------------------------------

[{'address': [u'Angel, London'],

... 30 items...

# Requests ------------------------------------------------

[<GET http://web:9312/properties/index_00001.html>]
```

正如期望的那样，parse() 返回了30 个Item 以及一个前往下一索引页的Request 。请使用scrapy parse 随意试验，比如传输--

depth=2 。

5.4　基于Excel文件爬取的爬虫

大多数情况下，每个源网站只会有一个爬虫；不过在某些情况下，你想要抓取的数据来自多个网站，此时唯一变化的东西就是所使用的XPath表达式。对于此类情况，如果为每个网站都使用一个爬虫则显得有些小题大做。那么可以只使用一个爬虫来爬取所有这些网站吗？答案是肯定的。

让我们为该实验创建一个新的爬虫，因为这次爬取的条目会和之前区别很大（实际上我们还没有在该项目中定义任何东西！）。假设此时在ch05 下的properties 目录中。让我们向上一层，如下面的代码所示进行操作。

```
$ pwd

/root/book/ch05/properties

$ cd ..

$ pwd
```

```
/root/book/ch05
```

我们创建了一个名为generic 的新项目，以及一个名为fromcsv 的爬虫。

```
$ scrapy startproject generic

$ cd generic

$ scrapy genspider fromcsv example.com
```

现在，创建一个.csv文件，其中包含想要抽取的信息。可以使用一个电子表格程序，比如Microsoft Excel，来创建这个.csv文件。填入如图5.5所示的几个URL和XPath表达式，然后将其命名为todo.csv，保存到爬虫目录当中（scrapy.cfg所在目录）。要想保存为.csv文件，需要在保存对话框中选择CSV文件（Windows）作为文件格式。

	A	B	C
1	url	name	price
2	http://web:9312/static/a.html	//*[@id="itemTitle"]/text()	//*[@id="prcIsum"]/text()
3	http://web:9312/static/b.html	//h1/text()	//span/strong/text()
4	http://web:9312/static/c.html	//*[@id="product-desc"]/span/text()	

图5.5　包含URL和XPath表达式的todo.csv

很好！如果一切都已就绪，你就可以在终端上看到该文件。

```
$ cat todo.csv

url,name,price

a.html,"//*[@id=""itemTitle""]/text()","//*[@id=""prcIsum""]/text()"

b.html,//h1/text(),//span/strong/text()
```

```
c.html,"//*[@id=""product-desc""]/span/text()"
```

Python有一个用于处理.csv 文件的内置库。只需通过import csv导入模块，然后就可以使用如下这些直截了当的代码，以字典的形式读取文件中的所有行了。在当前目录下打开Python提示符，就可以尝试如下代码。

```
$ pwd

/root/book/ch05/generic2

$ python

>>> import csv

>>> with open("todo.csv", "rU") as f:
```

```
    reader = csv.DictReader(f)

    for line in reader:

        print line
```

文件中的第一行会被自动作为标题行处理，并且会根据它们得出字典中键的名称。在接下来的每一行中，会得到一个包含行内数据的字典。我们使用for循环迭代每一行。当运行前面的代码时，可以得到如下输出。

```
{'url': ' http://a.html', 'price': '//*[@id="prcIsum"]/text()',
'name': '//*[@id="itemTitle"]/text()'}
{'url': ' http://b.html', 'price': '//span/strong/text()', 'name': '//
h1/text()'}
{'url': ' http://c.html', 'price': '', 'name': '//*[@id="product-
desc"]/span/text()'}
```

非常好。现在，可以编辑generic/spiders/fromcsv.py 这个爬虫了。我们将会用到.csv 文件中的URL，并且不希望有任何域名限制。因此，首先要做的事情就是移除start_urls 以及allowed_domains ，然后读取.csv 文件。

由于我们事先并不知道想要起始的URL，而是从文件中读取得到的，因此需要实现一个start_requests() 方法。对于每一行，创建Request ，然后对其进行yield 操作。此外，还会在reqeust.meta 中存储来自csv 文件的字段名称和XPath表达式，以便在parse() 函数中使用它们。然后，使用Item 和ItemLoader 填充Item 的字段。下面是完整的代码。

```
import csv
import scrapy
from scrapy.http import Request
from scrapy.loader import ItemLoader
from scrapy.item import Item, Field

class FromcsvSpider(scrapy.Spider):
    name = "fromcsv"

def start_requests(self):
    with open("todo.csv", "rU") as f:
        reader = csv.DictReader(f)
        for line in reader:
            request = Request(line.pop('url'))
            request.meta['fields'] = line
            yield request

def parse(self, response):
    item = Item()
    l = ItemLoader(item=item, response=response)
    for name, xpath in response.meta['fields'].iteritems():
        if xpath:
        item.fields[name] = Field()
```

```
        l.add_xpath(name, xpath)
return l.load_item()
```

接下来开始爬取，并将结果输出到out.csv 文件中。

```
$ scrapy crawl fromcsv -o out.csv

INFO: Scrapy 0.0.3 started (bot: generic)

...

DEBUG: Scraped from <200 a.html>

{'name': [u'My item'], 'price': [u'128']}

DEBUG: Scraped from <200 b.html>
```

```
{'name': [u'Getting interesting'], 'price': [u'300']}

DEBUG: Scraped from <200 c.html>

{'name': [u'Buy this now']}

...

INFO: Spider closed (finished)

$ cat out.csv

price,name

128,My item
```

```
300,Getting interesting

,Buy this now
```

正如爬取得到的结果一样，非常简洁直接！

在代码中，你可能已经注意到了几个事情。由于我们没有为该项目定义系统范围的Item，因此必须像如下代码这样手动为ItemLoader提供。

```
item = Item()
l = ItemLoader(item=item, response=response)
```

此外，我们还使用了Item的成员变量fields动态添加字段。为了能够动态添加新字段，并通过ItemLoader对其进行填充，需要实现的代码如下。

```
item.fields[name] = Field()
l.add_xpath(name, xpath)
```

最后，还可以使代码更加好看。硬编码todo.csv文件名不是一个非常好的实践。Scrapy提供了一个非常便捷的方法，用于传输参数到爬虫当中。当传输一个命令行参数-a时（比如：-a variable=value），就会为我们设置一个爬虫属性，并且可以通过self.variable取得该值。为了检查变量，并在未提供该变量时使用默认值，可以使用Python的getattr()方法：getattr(self, 'variable', 'default')。总之，我们将原来的with open…语句替换为如下语句。

```
with open(getattr(self, "file", "todo.csv"), "rU") as f:
```

现在，除非明确使用-a参数设置源文件名，否则将会使用todo.csv作为其默认值。当给出另一个文件another_todo.csv时，可以按如下方式运行。

```
$ scrapy crawl fromcsv -a file=another_todo.csv -o out.csv
```

5.5　本章小结

本章深入讨论了Scrapy爬虫的内部机制。我们学习了使用`FormRequest`进行登录，使用`Request/Response`的`meta`属性传输变量，使用相对XPath表达式和`Selector`，以及使用`.csv`文件作为源等。

接下来，第6章会讲解如何将爬虫部署到Scrapinghub云上，第7章将继续深入Scrapy的设置。

第6章　部署到Scrapinghub

在前面的几章中，我们了解了如何开发Scrapy爬虫。当我们对爬虫的功能感到满意时，接下来会有两个选项。如果我们需要的只是使用它们执行简单的抓取工作，那么此时使用开发机运行即可。而另一方面，更常见的情况是需要周期性地运行抓取任务，此时可以使用云服务器，如Amazon、RackSpace或其他提供商，不过这些都需要创建、配置和维护工作。此时就是Scrapinghub发挥作用的时候了。

Scrapinghub是Scrapy托管的Amazon服务器，它是由Scrapy开发者创建的Scrapy云基础设施提供商。它是一个付费服务，不过也提供了免费方案。如果你想在几分钟内，就能够让Scrapy爬虫运行在专业的创建和维护环境中的话，那么本章非常适合你。

6.1　注册、登录及创建项目

第一步是在`http://scrapinghub.com/`上面创建账号。我们所需填写的只有邮箱地址和密码。在单击确认邮件的链接后，就可以登录到其服务中。我们可以看到的第一个页面是个人面板。目前，我们还没有任何项目，因此现在单击**+Service** 按钮（1）来创建一个项目，如图6.1所示。

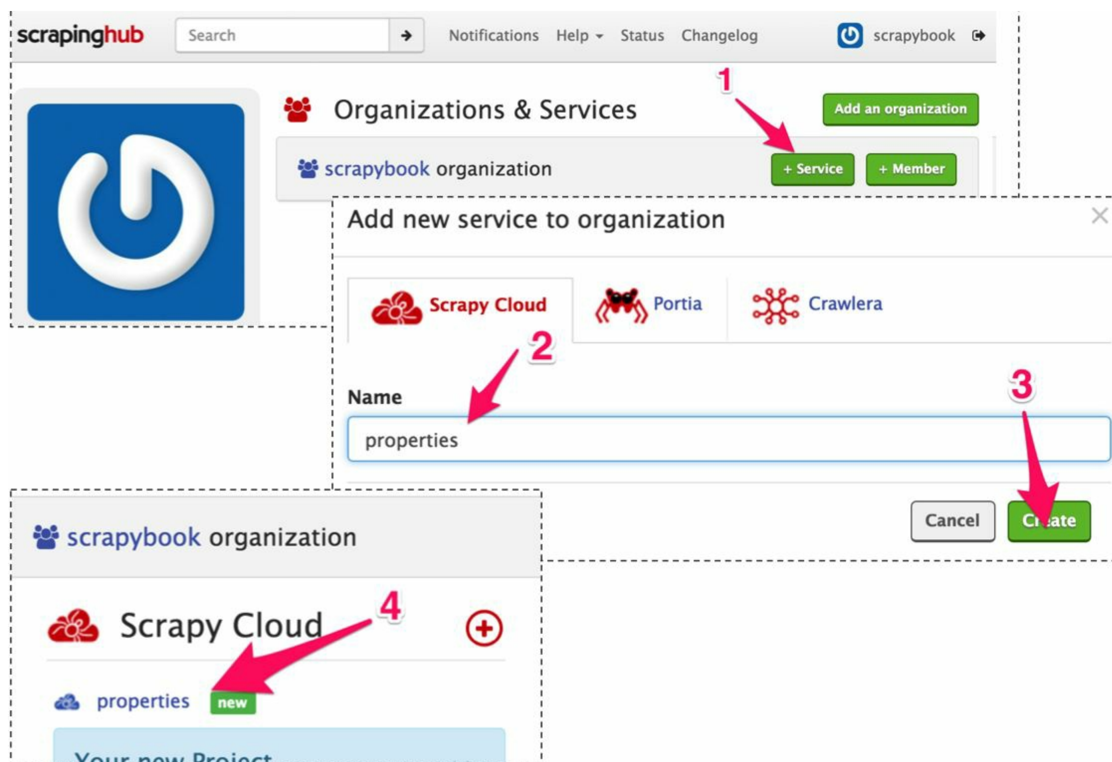


图6.1　在scrapinghub上创建新项目

将项目命名为`properties`（2），然后单击**Create**按钮（3）。之后，单击主页的**new**链接（4）打开该项目。

项目面板是项目中最重要的页面。在左侧的菜单中，可以看到几个区域，如图6.2所示。**Jobs**和**Spiders**区域分别提供关于运行和爬虫的信息。**Periodic Jobs**允许我们计划定期爬取任务。而另外4个区域目前来说对我们没有那么有用。

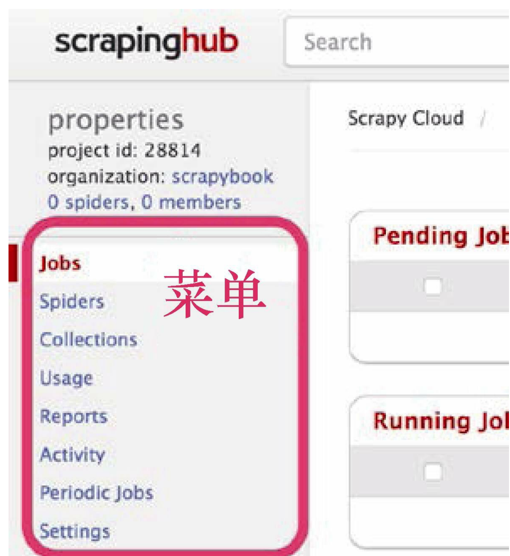


图6.2　主菜单

我们可以直接前往**Settings** 区域（1），如图6.3所示。与很多网站的设置不同，Scrapinghub的设置提供了很多功能，需要你十分了解它们。目前，我们的主要关注点是**Scrapy Deploy** 区域（2）。

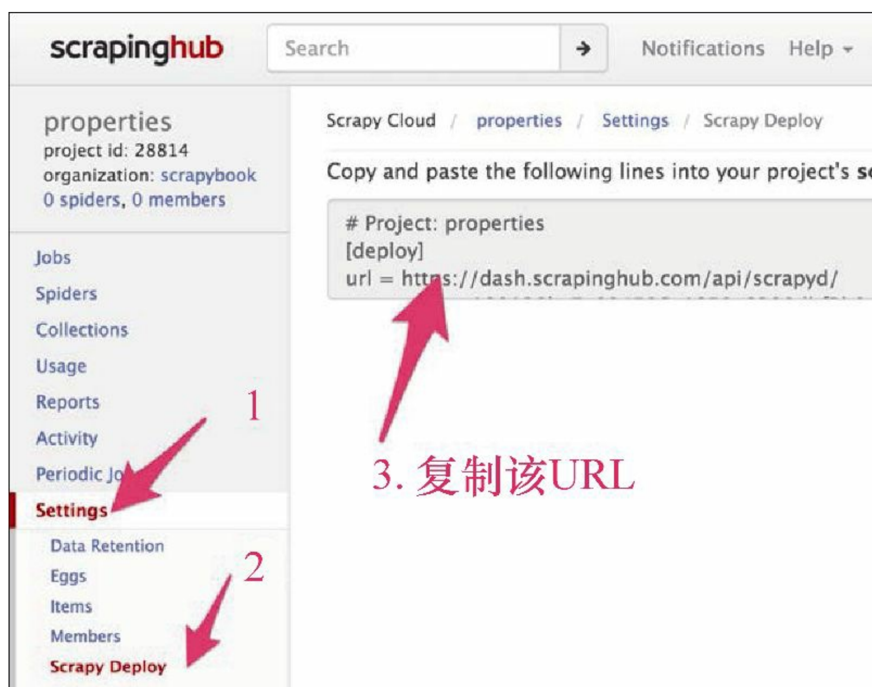


图6.3　爬虫部署设置

6.2　部署爬虫与计划运行

我们将直接从开发机进行部署。要想实现这一目标，只需将**Scrapy Deploy** 页面中的代码（3）拷贝到项目中的`scrapy.cfg` 文件中，替换掉默认的`[deploy]` 区域即可。你会注意到我们并不需要设置密码。我们将使用第4章中的房产项目作为示例，使用该爬虫的原因是目标数据需要能够在网络上访问到，和第4章使用的情况一样。在使用它之前，需要恢复原始的`settings.py` 文件，移除和Appery.io管道相关的引用。



本章代码在`ch06` 目录中。其中，该示例位于`ch06/properties` 目录中。

```
$ pwd

/root/book/ch06/properties

$ ls

```

```
properties scrapy.cfg

$ cat scrapy.cfg

...

[settings]

default = properties.settings

# Project: properties

[deploy]

url = http://dash.scrapinghub.com/api/scrapyd/
```

```
username = 180128bc7a0.....50e8290dbf3b0

password =

project = 28814
```

为了部署爬虫，还需要使用Scrapinghub提供的`shub`工具。可以通过`pip install shub`安装该工具，不过我们已经在开发机中已经安装好该工具了。可以使用下述方法登录Scrapinghub。

```
$ shub login

Insert your Scrapinghub API key : 180128bc7a0.....50e8290dbf3b0
```

```
Success.
```

我们已经将API key复制到`scrapy.cfg`文件中了，不过也可以通过单击Scrapinghub网站右上角的用户名，再单击**API Key**找到该值。无论如何，现在我们已经准备好使用`shub deploy`部署爬虫了。

```
$ shub deploy

Packing version 1449092838

Deploying to project "28814" in {"status": "ok", "project": 28814,

"version": "1449092838", "spiders": 1}

Run your spiders at: https://dash.scrapinghub.com/p/28814/
```

Scrapy将本项目中的所有爬虫打包，并上传到Scrapinghub当中。可以注意到，此时产生了两个新目录和一个新文件。这些只是辅助文件，如果不需要的话，可以安全地删除它们，不过通常情况下没必要在意它们。

```
$ ls

build project.egg-info properties scrapy.cfgsetup.py

$ rm -rf build project.egg-info setup.py
```

现在，当单击Scrapinghub的**Spiders** 区域（1）时，可以找到刚刚部

署的**tomobile** 爬虫，如图6.4所示。

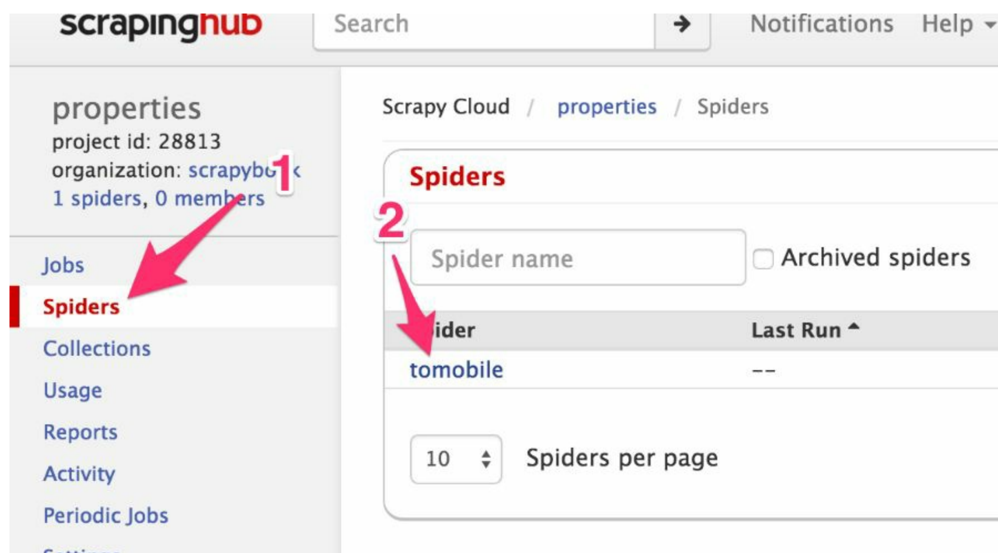


图6.4　选择爬虫

当单击它时（2），会进入到爬虫面板，如图6.5所示。该面板中包含大量信息，不过目前我们需要做的就是单击右上角的**Schedule** 按钮（3），然后在弹出的对话框中再次单击**Schedule** 按钮（4）。

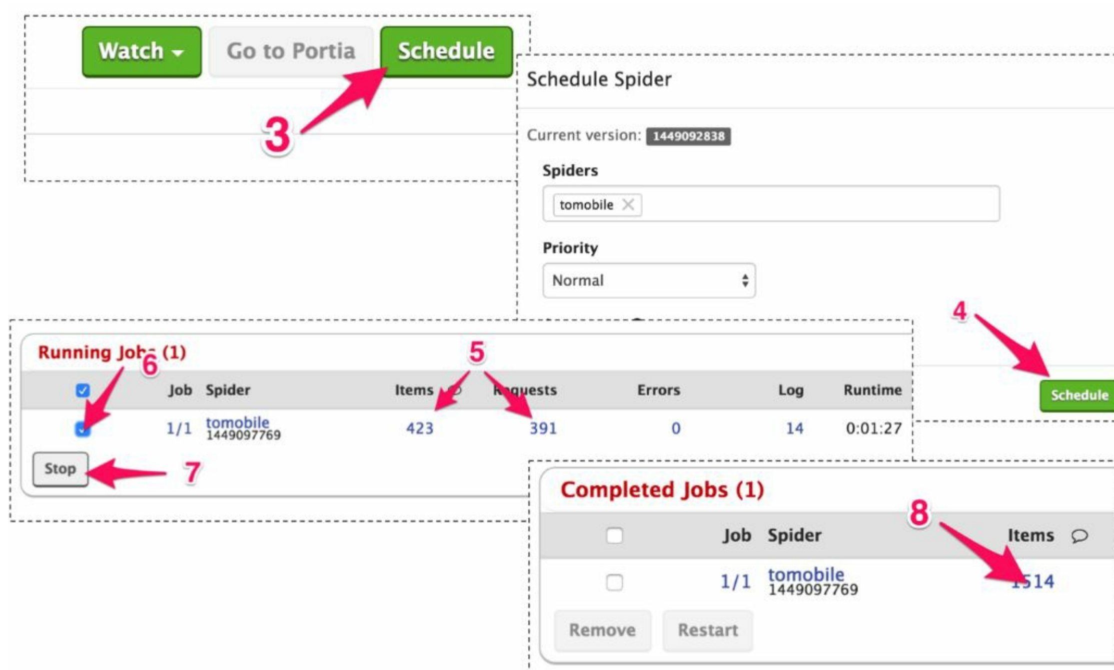


图6.5　计划爬虫运行

几秒钟之后，可以在页面中的**Running Jobs** 区域看到新的一行，之后**Requests** 和**Items** 的数值（5）开始不断增长。



与开发时的运行速度相比，此时的运行速度可能不会降低。Scrapinghub使用了算法预估每秒的请求数，能够让你在执行时不会被屏蔽。

让它运行一会儿，然后选择该任务的复选框（6），并单击**Stop** 按钮（7）。

几秒钟之后，我们的任务将会停止，并进入**Completed Jobs** 区域。要想查看已经抓取的条目，可以单击items链接中的数字（8）。

6.3　访问item

现在，我们需要前往任务页，如图6.6所示。在该页中，可以查看到我们的item（9），并确保其没有问题。我们还可以使用上面的控件进行过滤。当向下滚动页面时，更多的item会被自动加载出来。

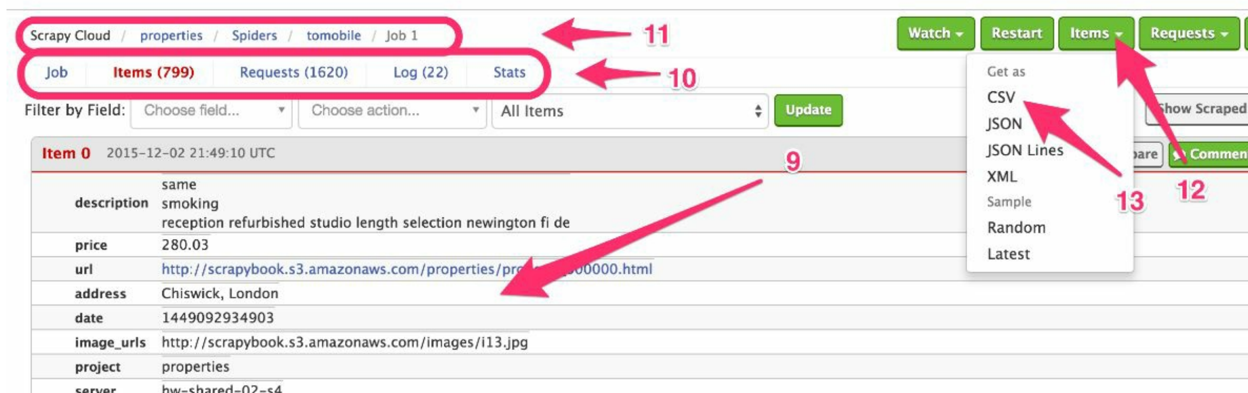


图6.6　查看及导出item

如果存在一些没能正常运行的情况，可以在**Items** 上方的**Requests** 和**Log** 中找到有用的信息（10）。可以使用顶部的面包屑导航回到爬虫或项目中（11）。当然，也可以通过单击左上方的**Items** 按钮（12），选择合适的选项（13），将item以常见的CSV、JSON、JSON行等格式下载下来。

另一种访问item的方式是通过Scrapinghub提供的Items API。我们所需做的就是查看任务或items页面中的URL，类似于下面这样。

```
https://dash.scrapinghub.com/p/28814/job/1/1/
```

在该URL中，**28814** 是项目编号（之前在`scrapy.cfg` 文件中设置

过该值），第一个1 是该爬虫的编号/ID（即"**tomobile** "爬虫），而第二个1则是任务编号。以上述顺序使用这3个数值，并使用我们的用户名/API Key进行验证，就可以在控制台中使用curl 建立到https://storage.scapinghub.com/ items/<project id>/<spider id>/<job id> 的请求，获取item，该过程如下所示。

```
$ curl -u 180128bc7a0.....50e8290dbf3b0: https://storage.scrapinghub.com/

items/28814/1/1

{"_type":"PropertiesItem","description":["same\r\nsmoking\r\nr...

{"_type":"PropertiesItem","description":["british bit keep eve...

...
```

如果它请求输入密码，我们将其留空即可。允许编程访问数据的特性使得我们可以编写应用，使用Scrapinghub作为数据存储后端。不过需要注意的是，这些数据并不是无限期存储的，而是依赖于订阅方案中的存储时间限制（对于免费方案来说该限制为7天）。

6.4 计划定时爬取

现在当你听到计划定时爬取任务只需要单击几下鼠标的话，应该不会再感到惊讶了。

该过程如图6.7所示。我们只需要前往**Periodic Jobs** 区域（1），单击**Add** （2），设置爬虫（3），调整爬取频率（4），最后单击**Save** 即可（5）。

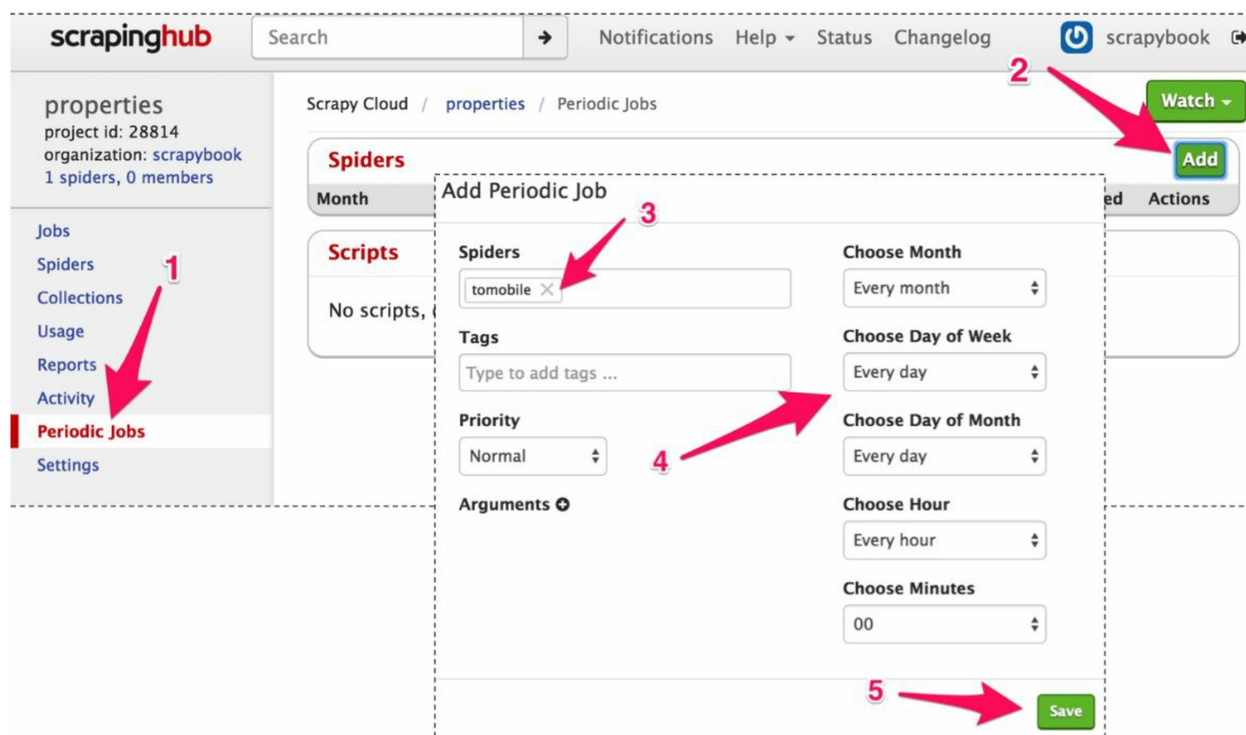


图6.7　计划定时爬取

6.5　本章小结

在本章中，我们拥有了第一次部署Scrapy项目的经验，这里我们使用了Scrapinghub将其部署到云端。我们计划运行任务，收集上千个item，并且可以通过使用API的方式非常容易地浏览和抽取它们。在接下来的章节中，我们将会继续提高知识水平，为自己创建一个类似Scrapinghub的小型服务器。首先，我们会在下一章中学习配置和管理。

第7章　配置与管理

前面章节讲解了使用Scrapy开发一个简单爬虫，并用它从网络上抽取数据是多么简单。Scrapy包含很多工具和功能，可以通过设置使它们可用。对于许多软件框架来说，设置是“令人讨厌的东西”，因为它需要根据系统如何运转进行调整。而对于Scrapy来说，设置则是其最重要的基本机制之一，除了调优和配置外，还可以启用功能，以及允许我们扩展框架。我们不打算与优秀的Scrapy文档竞争，只想辅助你更快地浏览设置概况，并找出与你最相关的内容。当你准备在生产环境中进行变更之前，请仔细阅读Scrapy文档。

7.1　使用Scrapy设置

在Scrapy中，可以按照5个递增的优先级修改设置。我们将会依次看到这 5 个等级。第一级是默认设置，通常不需要修改它，不过 `scrapy/settings/default_settings.py`（在系统的Scrapy源代码或Scrapy的GitHub中可以找到）中的代码确实值得一读。默认设置在命令级别中得以优化。实际上，除非想要实现自定义命令，否则无需考虑它。通常情况下，我们只会在命令级别下一级的项目 `<project_name>/settings.py` 文件中修改设置。这些设置只应用于当前项目。该级别最为方便，因为当我们将项目部署到云服务时，`settings.py` 文件将会打包在其中，并且由于它是一个文件，因此可以使用自己喜欢的文本编辑器轻松调整几十个设置。接下来一级是

每个爬虫的设置。通过在爬虫定义中使用`custom_settings`属性，可以轻松地为每个爬虫自定义设置。比如，可以通过该设置为一个指定的爬虫启用或禁用Item管道。最后，对于一些临时修改，可以使用命令行参数`-s`，在命令行中传输设置。我们在前面已经使用过几次，比如`-s CLOSESPIDR_PAGECOUNT=3`，即用于启用爬虫关闭扩展，以便爬虫尽早关闭。在该级别中，我们可能会去设置API secrets、密码等。不要将这些信息写入`settings.py`文件中，因为你不会希望它们意外出现在某些公开代码库当中。

在本节中，我们将会研究一些非常重要的常用设置。为了感受不同类型，可以在任意项目中尝试如下命令。

```
$ scrapy settings --get CONCURRENT_REQUESTS
16
```

你得到的是其默认值。然后，修改项目中的`<project_name>/settings.py`文件，为`CONCURRENT_REQUESTS`设置一个值，比如14。此时，前面的`scrapy settings`命令将会给出你

刚刚设置的那个值，之后不要忘记恢复该值。接下来，尝试从命令行中显式设置该参数，将会得到如下结果。

```
$ scrapy settings --get CONCURRENT_REQUESTS -s CONCURRENT_REQUESTS=19

19
```

前面的输出提示了一个很有意思的事情。`scrapy cwarl` 和`scrapy settings` 都只是命令。每个命令都能使用刚才描述的加载设置的方法，其示例如下所示。

```
$ scrapy shell -s CONCURRENT_REQUESTS=19

>>> settings.getint('CONCURRENT_REQUESTS')

19
```

当需要找出项目中某个设置的有效值时，可以使用前面给出的任意一种方法。现在，我们需要更加仔细地了解Scrapy的设置。

7.2　基本设置

Scrapy包含非常多的设置，因此为其分类成为了一个迫切的需求。我们将会从图7.1中总结出的大部分基本设置开始讨论。通过它们了解重要的系统特性，并且我们还将频繁地调整它们。

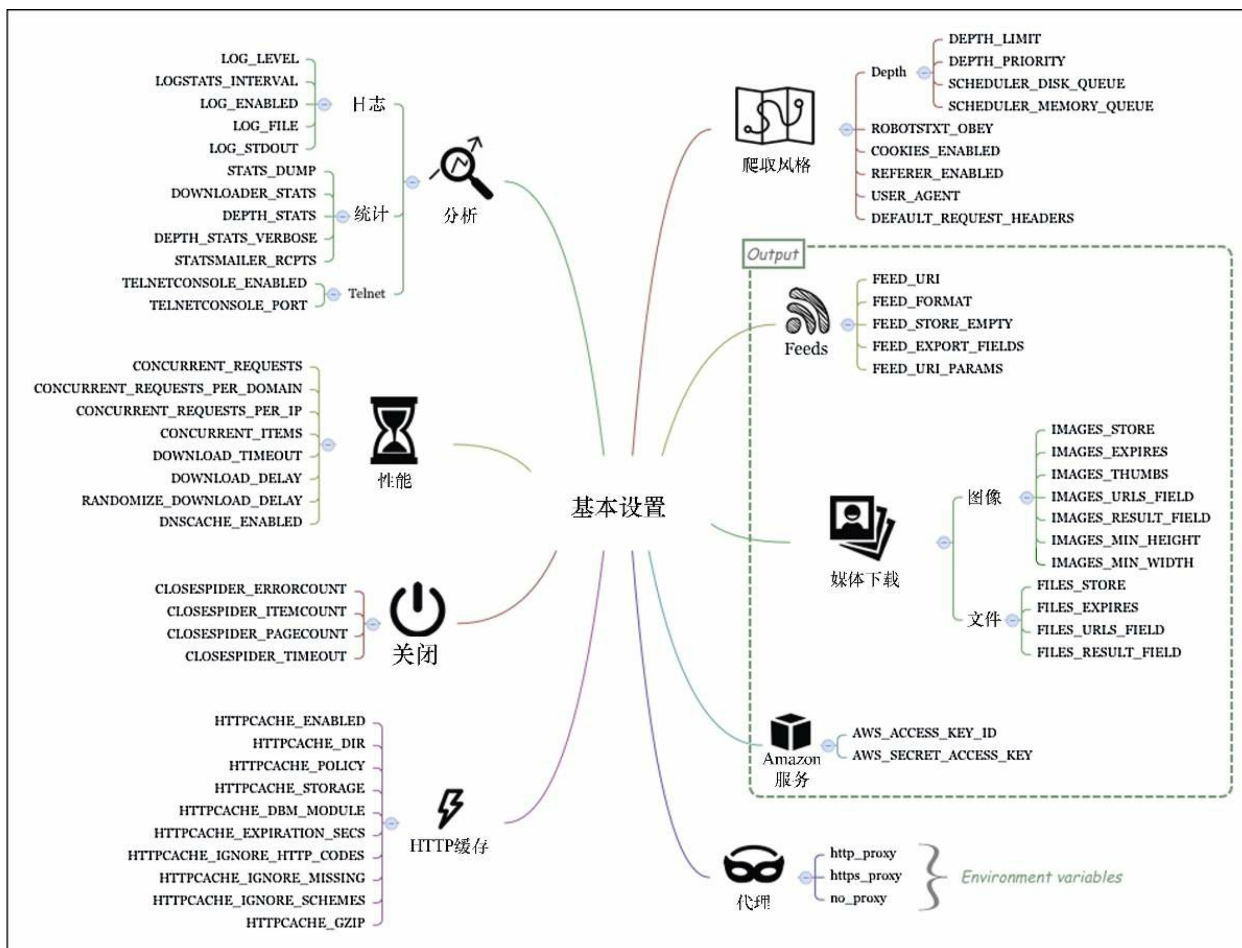


图7.1　Scrapy基本设置

7.2.1　分析

使用这些设置，你可以配置Scrapy，使其通过日志、统计和Telnet工具提供性能和调试信息。

1．日志

Scrapy基于严重程度，拥有不同的日志等级：`DEBUG`（最低等级）、`INFO`、`WARNING`、`ERROR`及`CRITICAL`（最高等级）。除此之外，还有一个`SILENT`等级，使用它将不记录任何日志。通过将`LOG_LEVEL`设置为希望日志记录的最低级别，可以限制日志文件只接

受指定等级以上的日志。我们一般将该值设为INFO，因为DEBUG级别过于详细。一个非常有用的Scrapy扩展是Log Stats扩展，该扩展会打印出每分钟抓取的item和页面的数量。日志频率使用LOGSTATS_INTERVAL进行设置，其默认值为60秒。该设置的频率过低，所以在我开发时，会将该值设置为5秒，因为大多数运行都很短暂。写入日志的文件可以通过LOG_FILE设置。除非将LOG_ENABLED的值设置为False进行显式禁用，否则日志将会输出到标准错误当中。最后，可以通过设置LOG_STDOUT为True，告知Scrapy将所有标准输出（比如："print"消息）写入日志。

2．统计

STATS_DUMP默认是开启的，它会在爬虫结束运行时，将统计信息收集器中的值转存到日志当中。可以通过将DOWNLOADER_STATS设置为False，控制是否为下载记录统计信息。还可以通过DEPTH_STATS设置，控制是否收集站点深度的统计信息。要想了解有关深度的更多细节，可以将DEPTH_STATS_VERBOSE设为True。STATSMAILER_RCPTS是一个邮件列表（比如设置为['my@mail.com']），当爬取完成时，会向该列表中的邮箱发送邮件。无需经常调整这些设置，不过它们偶尔会在调试时帮助到你。

3．Telnet

Scrapy包含一个内置的Telnet控制台，可以为你提供正在运行中的Scrapy进程的Python shell。TELNETCONSOLE_ENABLED默认情况下是开启的，而TELNETCONSOLE_PORT决定了连接到控制台的端口。你可能需要修改该值，以防止端口冲突。

示例1——使用Telnet

在某些情况下，需要查看正在运行的Scrapy的内部状态。下面让我们看看如何使用Telnet控制台完成该操作。



本章代码位于ch07 目录中。其中，本示例在ch07/properties 目录中。

```
$ pwd

/root/book/ch07/properties

$ ls

properties scrapy.cfg
```

使用如下命令开始爬取。

```
$ scrapy crawl fast

...

[scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023:6023
```

上面的消息意味着Telnet已经被激活，并且使用6023端口进行监听。现在，可以在另一个终端中，使用telnet命令连接它。

```
$ telnet localhost 6023

>>>
```

此时，该控制台会提供一个Scrapy内部的Python控制台。你可以查看某些组件，比如使用engine 变量查看引擎，不过为了能够更快地了解状态概况，可以使用est() 命令。

```
>>> est()

Execution engine status

time()-engine.start_time           : 5.73892092705

engine.has_capacity()              : False

len(engine.downloader.active)       : 8

...
```

```
len(engine.slot.inprogress)        : 10

...

len(engine.scraper.slot.active)    : 2
```

第10章将会探讨其中的一些度量标准。此时将发现你依然是在Scrapy引擎内部运行它。假设使用了如下命令：

```
>>> import time

>>> time.sleep(1) # Don't do this!
```

此时，你会发现在另一个终端中会出现短暂的暂停。显然，该控制台不是用来计算Pi值前100万位的合适地点。你可以在该控制台中操作的事情还包括暂停、继续和终止爬取。你可能会发现，在远程机器操作Scrapy会话时，这些事情和终端通常都很有用。

```
>>> engine.pause()

>>> engine.unpause()

>>> engine.stop()

Connection closed by foreign host.
```

7.2.2 性能

第10章将会详细介绍关于性能的设置，这里仅作为一个小结。性能设置可以让我们根据特定的工作负载调整Scrapy的性能特性。CONCURRENT_REQUESTS 用于设置同时执行的最大请求数。大多数情况下，该设置用于防止在爬取不同网站（域名/IP）时超出服务器出站容量。除此之外，还可以找到更加严格的 CONCURRENT_REQUESTS_PER_DOMAIN 以及 CONCURRENT_REQUESTS_PER_IP 。这两个设置分别通过限制同时对每个域名或 IP 地址发出的请求数，达到保护远程服务器的效果。当CONCURRENT_REQUESTS_PER_IP 为非零值时，CONCURRENT_REQUESTS_PER_DOMAIN 就会被忽略。这些设置不是以秒为单位的。如果CONCURRENT_REQUESTS = 16 ，而请求平均花费1/4秒的话，你的限制就是每秒16/0.25 = 64个请求。CONCURRENT_ITEMS 用于设置对每个响应同时处理的最大item数量。你可能会发现该设置并没有它看起来那么实用，因为很多情况下，每个页面或请求中只有一个Item 。并且，其默认值100也比较随意。如果减小该值，比如减小到10或者1，你甚至可能会看到性能提升，这取决于每个请求的Item数量，以及管道的复杂程度。还需要注意的是，由于该值是每个请求时的数量，如果限制了CONCURRENT_REQUESTS = 16 、CONCURRENT_ITEMS = 100 ，那么可能意味着会有1600个item同时在尝试写入数据库。一般来说，建议将该值设置得更保守一些。

对于下载，DOWNLOAD_TIMEOUT 决定了下载器在取消一个请求之前需要等待的时间，其默认值为180秒，这似乎有些偏高（当并发请求数为16时，这意味着站点下载的速度大约为5页/分钟）。建议降低该值，

比如当存在超时问题时，将其降低为10秒。默认情况下，Scrapy将两次下载间的延迟设置为0，以最大化抓取速度。可以使用DOWNLOAD_DELAY设置将其修改为更加保守的下载速度。有些网站会将请求频率作为“机器人”行为的测量指标。通过设置DOWNLOAD_DELAY，还会在下载延迟中启用一个±50%的随机偏移量。可以通过将RANDOMIZE_DOWNLOAD_DELAY设置为False来禁用该功能。

最后，为了更快的DNS查找，Scrapy默认使用了DNSCACHE_ENABLED设置，启用了内存中的DNS缓存。

7.2.3　提前终止爬取

Scrapy的CloseSpider扩展可以在达成某个条件时，自动终止爬虫爬取。可以分别使用CLOSESPIDER_TIMEOUT（以秒计）、CLOSESPIDER_ITEMCOUNT、CLOSESPIDER_PAGECOUNT以及CLOSESPIDER_ERRORCOUNT这些设置，配置在一段时间后、抓取一定数量item后、接收到一定数量响应后或是遇到一定数量错误后，关闭爬虫。通常情况下，你会在运行爬虫时使用命令行的方式设置这些内容，我们已经在前面的几章中做过几次此类操作。

```
$ scrapy crawl fast -s CLOSESPIDER_ITEMCOUNT=10



$ scrapy crawl fast -s CLOSESPIDER_PAGECOUNT=10
```

```
$ scrapy crawl fast -s CLOSESPIDER_TIMEOUT=10
```

7.2.4 HTTP缓存和离线运行

Scrapy的HttpCacheMiddleware组件（默认未激活）为HTTP请求和响应提供了一个低级的缓存。当启用该组件时，缓存会存储每个请求及其对应的响应。通过将HTTPCACHE_POLICY设置为scrapy.contrib.httpcache.RFC2616Policy，可以启用一个遵从RFC2616的更复杂的缓存策略。为了启用该缓存，还需要将HTTPCACHE_ENABLED设置为True，并将HTTPCACHE_DIR设置为文件系统中的一个目录（使用相对路径将会在项目的数据文件夹下创建一个目录）。

还可以选择通过设置存储后端类HTTPCACHE_STORAGE为scrapy.contrib.httpcache.DbmCacheStorage，为缓存文件指定数据库后端，并且还可以选择调整HTTPCACHE_DBM_MODULE设置（默认为任意数据库管理系统）。还有一些设置可以用于缓存行为调优，不过默认值已经能够为你很好地服务了。

示例2——使用缓存的离线运行

假设你运行了如下代码：

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=5000
```

你会发现大约一分钟后运行可以完成。如果此时无法访问Web服务器，可能就无法爬取任何数据。假设你现在使用如下代码，再次运行爬虫。

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=5000 -s

HTTPCACHE_ENABLED=1

...

INFO: Enabled downloader middlewares:...*HttpCacheMiddleware*
```

你会注意到此时启用了HttpCacheMiddleware，当查看当前目录下的隐藏目录时，将会发现一个新的.scrapy目录，目录结构如下所示。

```
$ tree .scrapy | head

.scrapy

└── httpcache

    └── easy

        ├── 00
```

```
│   ├── 002054968919f13763a7292c1907caf06d5a4810
│   │   ├── meta
│   │   ├── pickled_meta
│   │   ├── request_body
```

```
│    │    ├── request_headers
│    │    ├── response_body
...
```

现在，如果重新运行爬虫，获取略少于前面数量的item时，就会发现即使在无法访问Web服务器的情况下，也能完成得更加迅速。

```
$ scrapy crawl fast -s LOG_LEVEL=INFO -s CLOSESPIDER_ITEMCOUNT=4500 -s
```

```
HTTPCACHE_ENABLED=1
```

我们使用了略少于前面数量的item作为限制，是因为当使用CLOSESPIDER_ITEMCOUNT结束时，一般会在爬虫完全结束前读取更多的页面，但我们不希望命中的页面不在缓存范围内。要想清理缓存，只需删除缓存目录即可。

```
$ rm -rf .scrapy
```

7.2.5 爬取风格

Scrapy允许我们调整选择优先爬取页面的方式。可以在DEPTH_LIMIT设置中设定最大深度，该值为0时表示不限制。通过DEPTH_PRIORITY设置，可以基于请求的深度指定优先级。最值得注意的是，可以将该值设置为正数，以执行广度优先爬取，并将任务队列由

LIFO（后入先出）转为FIFO（先入先出）：

```
DEPTH_PRIORITY = 1

SCHEDULER_DISK_QUEUE = 'scrapy.squeue.PickleFifoDiskQueue'

SCHEDULER_MEMORY_QUEUE = 'scrapy.squeue.FifoMemoryQueue'
```

在爬取时进行这些设置非常有用，比如，在一个新闻门户网站中，最近的新闻更应该接近首页，并且每个新闻页都有到其他相关新闻的链接。Scrapy的默认行为是对首页的前几个新闻报道进行尽可能深地爬取，之后才会继续爬取接下来的头版新闻。而广度优先的顺序则是首先爬取最顶层的新闻，之后才会进一步深入，当结合`DEPTH_LIMIT` 设置时，比如设为3，可以让你快速浏览门户网站中最近的新闻。

网站在其根目录下使用Web标准的`robots.txt` 文件，声明它们允许的爬取策略，以及不希望被访问的网站结构。如果将`ROBOTSTXT_OBEY` 设置为`True` ，Scrapy将会遵守该约定。如果启用了

该设置，请在调试时记住该点，以防发现任何意外的行为。

CookiesMiddleware 显然包含了和cookie 相关的所有操作，其中包括会话跟踪、准许登录等。如果你想拥有更“私密”的爬取，可以通过将COOKIES_ENABLED 设置False 以禁用。禁用cookie还会轻微降低你使用的带宽，并且可能会对你的爬取操作有一点提速，当然它会依赖于你爬取的网站。与CookiesMiddleware 类似，REFERER_ENABLED 的默认设置是True ，即启用了用于填充Referer头的RefererMiddleware 。可以使用DEFAULT_REQUEST_HEADERS 自定义头部。你可能会发现该设置对于某些奇怪的网站很有用，在这些网站中只有包含了特定请求头的请求才不会被禁止。最后，自动生成的settings.py 文件推荐我们设置USER_AGENT 。该设置的默认值是Scrapy的版本，而我们需要将其修改为能够让网站拥有者联系到我们的信息。

7.2.6 feed

feed可以让你将Scrapy抓取得到的数据输出到本地文件系统或远程服务器当中。FEED_URI.FEED_URI 决定了feed的位置，该设置中可能会包含命名参数。比如，scrapy fast -o "%(name)s_%(time)s.jl" 将会自动以当前时间和爬虫名称（fast ）填充输出文件名。如果需要使用一个自定义参数，比如%(foo)s ，那么feed 输出器需要你在爬虫中提供foo 属性。此外，feed的存储，如S3、FTP或本地文件系统，也定义在URI中。例如，FEED_URI='s3://mybucket/file.json' 将使用你的Amazon凭证（AWS_ACCESS_KEY_ID 和AWS_SECRET_ACCESS_KEY ）上传文件到Amazon的S3当中。Feed的格式（JSON、JSON Line、CSV及XML）可

以使用FEED_FORMAT 确定。如果没有设定该设置，Scrapy将会根据FEED_URI 的扩展名猜测格式。通过将FEED_STORE_EMPTY 设置为True ，可以选择输出空的feed。此外，还可以使用FEED_EXPORT_FIELDS 设置，选择只输出指定的几个字段。该设置对于具有固定标题列的.csv 文件尤其有用。最后，FEED_URI_PARAMS 用于定义对FEED_URI 中任意参数进行后置处理的函数。

7.2.7　媒体下载

Scrapy可以使用图像管道下载媒体内容，此外还可以将图像转换为不同的格式、生成缩略图以及基于大小过滤图像。

IMAGES_STORE 设置用于设定图像存储的目录（使用相对路径时，将会在项目根目录下创建目录）。每个Item 的图像URL应该在image_urls 字段中设定（可以被IMAGES_URLS_FIELD 设置覆写），而下载图像的文件名则是在一个新的images 字段中设定（可以被IMAGES_RESULT_FIELD 设置覆写）。可以使用IMAGES_MIN_WIDTH 和IMAGES_MIN_HEIGHT 设置过滤过小的图像。IMAGES_EXPIRES 决定了图像在过期前保留在缓存中的天数。对于缩略图的生成，可以使用IMAGES_THUMBS 设置，它可以让你按照一种或多种尺寸生成缩略图。比如，可以让Scrapy生成一种图标大小的缩略图以及一种用于每次图像下载时的中等大小缩略图。

1．其他媒体

可以使用文件管道下载其他媒体文件。与图像类似，FILES_STORE 设置用于确定已下载文件的存放位置，而FILES_EXPIRES 设置用于确

定文件保留的天数。FILES_URLS_FIELD 以及FILES_RESULT_FIELD 设置都和对应的IMAGES_* 设置的功能相似。文件管道和图像管道可以同时激活，不会产生冲突。

示例3——下载图像

为了能够使用图像功能，必须使用sudo pip install image 安装图像包。在我们的开发机中，已经为大家安装好该三方包了。想要启用图像管道，只需要编辑项目的settings.py 文件，添加少量设置。首先，需要在ITEM_PIPELINES 中包含scrapy.pipelines.images.ImagesPipeline 。然后，设置IMAGES_STORE 为相对路径"images" ，此外还可以选择通过IMAGES_THUMBS 设置一些缩略图的描述，相关代码如下所示。

```
ITEM_PIPELINES = {
...
    'scrapy.pipelines.images.ImagesPipeline': 1,
}
IMAGES_STORE = 'images'
IMAGES_THUMBS = { 'small': (30, 30) }
```

我们在Item中已经包含了合适的image_urls 字段，所以现在可以参照如下命令执行爬虫了。

```
$ scrapy crawl fast -s CLOSESPIDER_ITEMCOUNT=90
```

```
...

DEBUG: Scraped from <200 http://http://web:9312/.../index_00003.html/

property_000001.html>{

   'image_urls': [u'http://web:9312/images/i02.jpg'],

   'images': [{'checksum': 'c5b29f4b223218e5b5beece79fe31510',

              'path': 'full/705a3112e67...a1f.jpg',

              'url': 'http://web:9312/images/i02.jpg'}],

...
```

```
$ tree images

images

├── full

│   ├── 0abf072604df23b3be3ac51c9509999fa92ea311.jpg

│   ├── 1520131b5cc5f656bc683ddf5eab9b63e12c45b2.jpg

...

└── thumbs



└── small
```

```
├── 0abf072604df23b3be3ac51c9509999fa92ea311.jpg
├── 1520131b5cc5f656bc683ddf5eab9b63e12c45b2.jpg
...
```

可以看到图像成功下载，并且创建了缩略图。主文件的JPG名称按照预期存储在了`images`字段当中，因此很容易推测缩略图的路径。如果想要清空图像，我们可以使用`rm -rf images`。

7.2.8 Amazon Web服务

Scrapy对访问Amazon Web服务有内置支持。你可以在AWS*ACCESS* KEY_ID设置中存储AWS访问密钥，在AWS_SECRET_ACCESS_KEY设置中存储私密密钥。默认情况下，这些设置均为空。可以在如下场景中使用：

- 当下载以`s3://`开头的URL时（而不是https://等）；
- 当通过媒体管道使用`s3://`路径存储文件或缩略图时；
- 当在`s3://`目录中存储`Item`的输出Feed时。

不要将这些设置存储在`settings.py`文件当中，以防未来某天由于任何原因造成代码公开时被泄露。

7.2.9 使用代理和爬虫

Scrapy的`HttpProxyMiddleware`组件允许你使用代理设置，根据UNIX约定，这些设置是通过`http_proxy`、`https_proxy`以及`no_proxy`这几个环境变量定义的。该组件默认是启用状态的。

示例4——使用代理和Crawlera的智能代理

DynDNS（或任何类似的服务）提供了一个免费的在线工具，用于查看当前的IP地址。使用Scrapy shell，我们向`checkip.dyndns.org`发送请求，查看响应，获取当前的IP地址。

```
$ scrapy shell http://checkip.dyndns.org



>>> response.body
```

```
'<html><head><title>Current IP Check</title></head><body>Current IP

Address: xxx.xxx.xxx.xxx</body></html>\r\n'

>>> exit( )
```

想要开始代理请求，需要退出shell，并使用`export`命令设置新的代理。可以通过搜索HMA的公开代理列表测试免费代理（`http://proxylist. hidemyass.com`）。比如，我们从该列表中选择了一个IP为10.10.1.1、端口为80的代理（非真实存在的代理，请将其替换为你自己的代理地址），可以按照如下操作。

```
$ # First check if you already use a proxy
```

```
$ env | grep http_proxy

$ # We should have nothing. Now let's set a proxy

$ export http_proxy=http://10.10.1.1:80
```

按照刚才的步骤重新运行scrapy shell，可以看到执行的请求使用了不同的IP。此外，你还会发现这些代理通常速度都很慢，而且在一些情况下无法成功，如果遇到这类情况，可以尝试更换为其他的代理。如果想要禁用代理，则需要退出Scrapy shell，并执行`unset http_proxy`（或恢复为之前的值）。

Crawlera是Scrapinghub的一项服务，可以为Scrapy的开发者提供一个非常智能的代理。除了在后台使用很大的IP池路由你的请求外，该代理还会调整延迟和失败重试，让你在保持尽可能快的情况下，获得尽可能多且稳定的成功响应流。它基本上可以使爬虫开发者的梦想成真，并且只需像之前那样，设置`http_proxy`环境变量，就可以使用。

```
$ export http_proxy=myusername:mypassword@proxy.crawlera.com:8010
```

除了HTTP代理外，Crawlera还可以通过Scrapy的中间件组件方式使用。

7.3　进阶设置

现在，我们要探讨一些Scrapy中不太常见的方面，以及Scrapy扩展的相关设置，后续章节中会详细介绍这些内容。这些进阶设置如图7.2所示。

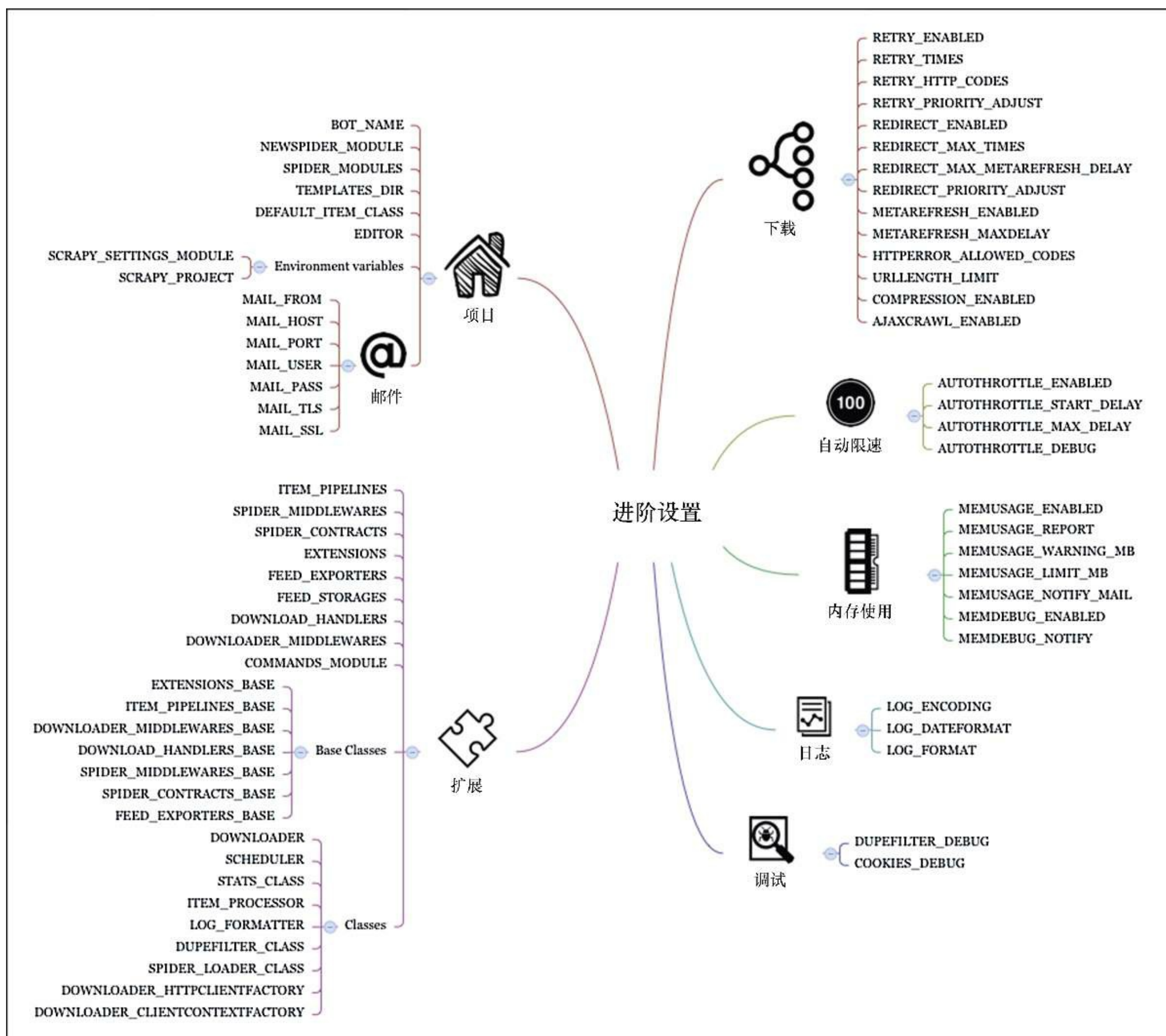


图7.2　Scrapy进阶设置

7.3.1　项目相关设置

在这里可以找到一些与具体项目相关的管理设置，如BOT_NAME 、SPIDER_MODULES 等。你可以快速浏览一下这些设置的文档，因为它们会提升具体用例的生产效率，不过通常情况下，Scrapy的 startproject 和genspider 命令都已经提供了合理的默认值，即使不显式修改它们，也能很好地运行。邮件相关的设置，比如MAIL_FROM ，

可以让你配置MailSender类，该类目前用于统计邮件信息（另外参见STATSMAILER_RCPTS）以及内存使用信息（另外参见MEMUSAGE_NOTIFY_MAIL）。还有两个环境变量：SCRAPY_SETTINGS_MODULE以及SCRAPY_PROJECT，可以让你调整Scrapy项目与其他项目集成的方式，比如Django项目。scrapy.cfg还允许你调整设置模块的名称。

7.3.2　Scrapy扩展设置

这些设置能够让你扩展并修改Scrapy的几乎所有方面。这些设置中最重要的当属ITEM_PIPELINES。它可以让你在项目中使用Item处理管道。第9章会看到更多的例子。除了管道之外，还可以通过不同的方式扩展Scrapy，其中一些将会在第8章中进行总结。COMMANDS_MODULE允许我们添加常用命令。比如，可以在properties/hi.py文件中添加如下内容。

```
from scrapy.commands import ScrapyCommand
class Command(ScrapyCommand):
    default_settings = {'LOG_ENABLED': False}
    def run(self, args, opts):
        print("hello")
```

当在settings.py文件中添加COMMANDS_MODULE='properties.hi'时，就激活了这个小命令，我们可以在Scrapy帮助中看到它，并且通过scrapy hi运行。在命令的default_settings中定义的设置，会被合并到项目的设置当中，并覆

盖默认值，不过其优先级低于settings.py 文件或命令行中设定的设置。

Scrapy使用-_BASE 字典（比如FEED_EXPORTERS_BASE ）存储不同框架扩展的默认值，并允许我们在settings.py 文件或命令行中，通过设置它们的非-_BASE 版本（比如FEED_EXPORTERS ）进行自定义。

最后，Scrapy使用DOWNLOADER 、SCHEDULER 等设置，保存系统基本组件的包/ 类名。我们可以继承默认的下载器（scrapy.core.downloader.Downloader ），重载一些方法，然后将DOWNLOADER 设置为自定义的类。这样可以让开发者大胆地对新特性进行实验，并且可以简化自动化测试过程，不过除非你明确了解自己做的事情，否则不要轻易修改这些设置。

7.3.3　下载调优

RETRY_* 、REDIRECT_* 以及METAREFRESH_* 设置分别用于配置重试、重定向以及元刷新中间件。例如，将REDIRECT_PRIORITY_ADJUST 设为2，意味着每次发生重定向时，新请求将会在所有非重定向请求完成服务后才会被调度；而将REDIRECT_MAX_TIMES 设置为20，则表示在执行20次重定向后，下载器将会放弃尝试，并返回目前所见到的内容。这些设置在爬取一些运行不太正常的网站时非常有用，不过在大多数情况下，默认值已经可以提供很好的服务了。它同样也适用于HTTPERROR_ALLOWED_CODES 以及URLLENGTH_LIMIT 。

7.3.4　自动限速扩展设置

AUTOTHROTTLE_* 设置用于启用并配置自动限速扩展。虽然对它有很大期望，但从实践来看，我发现它往往有些保守，不容易调整。它使用下载延迟，来了解我们和目标服务器的负载情况，并据此调整下载器的延迟。如果你很难找到DOWNLOAD_DELAY 的最佳值（默认为0），就会发现该模块很有用。

7.3.5　内存使用扩展设置

MEMUSAGE_* 设置用于启用并配置内存使用扩展。当超出内存限制时，将会关闭爬虫。当运行在共享环境时，该设置非常有用，因为此时需要非常礼貌的行为。大多数情况下，你可能会发现它只有在接收报警邮件时才会有用，此时我们需要将MEMUSAGE_LIMIT_MB 设置为0，禁用关闭爬虫的功能。该扩展只在类UNIX平台上适用。

MEMDEBUG_ENABLED 和MEMDEBUG_NOTIFY 用于启用并配置内存调试扩展，在爬虫关闭时打印出仍然存活的引用数量。总之，追踪内存泄露不是一件简单而有趣的事情（好吧，它还是有一些乐趣的）。我们可以阅读 *Debugging memory leaks with trackref* 这篇优秀的文档，了解更多内存泄露排查的方法，不过最重要的建议是，保持你的爬虫相对简短、批量处理，并且需要根据服务器的能力运行。我认为没有什么好的理由可以让我们批量运行超过几千页或几分钟。

7.3.6　日志和调试

最后，还有一些日志和调试功能。LOG_ENCODING 、LOG_DATEFORMAT 和LOG_FORMAT 可以用来调整日志格式，当准备使用日志管理解决方案时（比如Splunk、Logstash和Kibana），会发现这

些设置非常有用。DUPEFILTER_DEBUG和COOKIES_ DEBUG将会帮助你调试相对复杂的情况，比如得到的请求数少于预期或会话意外丢失。

7.4　本章小结

通过阅读本章，我相信与从头开始编写爬虫相比，你能体会到使用Scrapy功能所带来的深度和广度。如果你想调整或扩展Scrapy的功能，可以有很多选项，我们将会在下一章中看到它们。

第8章　Scrapy编程

到目前为止，我们编写的爬虫主要用于定义爬取数据源的方式以及如何从中抽取信息。除了爬虫外，Scrapy还提供了能够调整其大多数方面功能的机制。比如，你可能会发现自己经常在处理如下的一些问题。

1．你需要从同一个项目的其他爬虫中复制、粘贴大量代码。重复的代码与数据更加相关（比如，执行字段计算），而不是数据源。

2．你需要编写脚本，对`Item`进行后处理，执行像删除重复条目或后置处理值的事情。

3．你在不同的项目中有重复的代码，用于处理基础架构。比如，你可能需要登录并向专有仓库传输文件，向数据库中添加`Item`或在爬虫执行完成时触发后置处理操作。

4．你发现Scrapy的某个方面与你希望的功能并不完全一致，你想在自己的大部分项目中使用自定义或变通的方案。

Scrapy开发者所设计的架构，能够为我们解决这些常见的问题。我们将会在本章后续部分研究该架构。不过我们首先介绍支持Scrapy的引擎，该引擎叫作**Twisted**。

8.1　Scrapy是一个Twisted应用

Scrapy是一个内置使用了Python的Twisted框架的抓取应用。Twisted确实有些与众不同，因为它是事件驱动的，并且鼓励我们编写异步代码。习惯它需要一些时间，不过我们将通过只学习和Scrapy有关的部分，从而让任务变得相对简单一些。我们还可以在错误处理方面轻松一些。GitHub上的完整代码会有更加彻底的错误处理，不过在本书中将忽略该部分。

让我们从头开始。Twisted与众不同是因为它的主要口号。



在任何情况下，都不要编写阻塞的代码。

代码阻塞的影响很严重，而可能造成代码阻塞的原因包括：

- 代码需要访问文件、数据库或网络；
- 代码需要派生新进程并消费其输出，比如运行shell命令；
- 代码需要执行系统级操作，比如等待系统队列。

Twisted提供的方法允许我们执行上述所有操作甚至更多操作时，无需再阻塞代码执行。

为了展示两种方式的不同，我们假设有一个典型的同步抓取应用（见图8.1）。假设该应用包含4 个线程，并且在一个给定的时刻，其中3 个线程处于阻塞状态，用于等待响应，而另一个线程被阻塞，用于执行数据库写访问以保存`Item` 。在任何给定时刻，很有可能无法找到抓

取应用的一个执行其他事情的线程，只能等待一些阻塞操作完成。当阻塞操作完成时，一些计算操作可能占用几微秒，然后线程再次被阻塞，执行其他阻塞操作，这很可能持续至少几毫秒的时间。总体来说，服务器不会是空闲的，因为它运行了几十个应用程序，并使用了上千个线程，因此，在一些细致的调优后，CPU才能够合理利用。

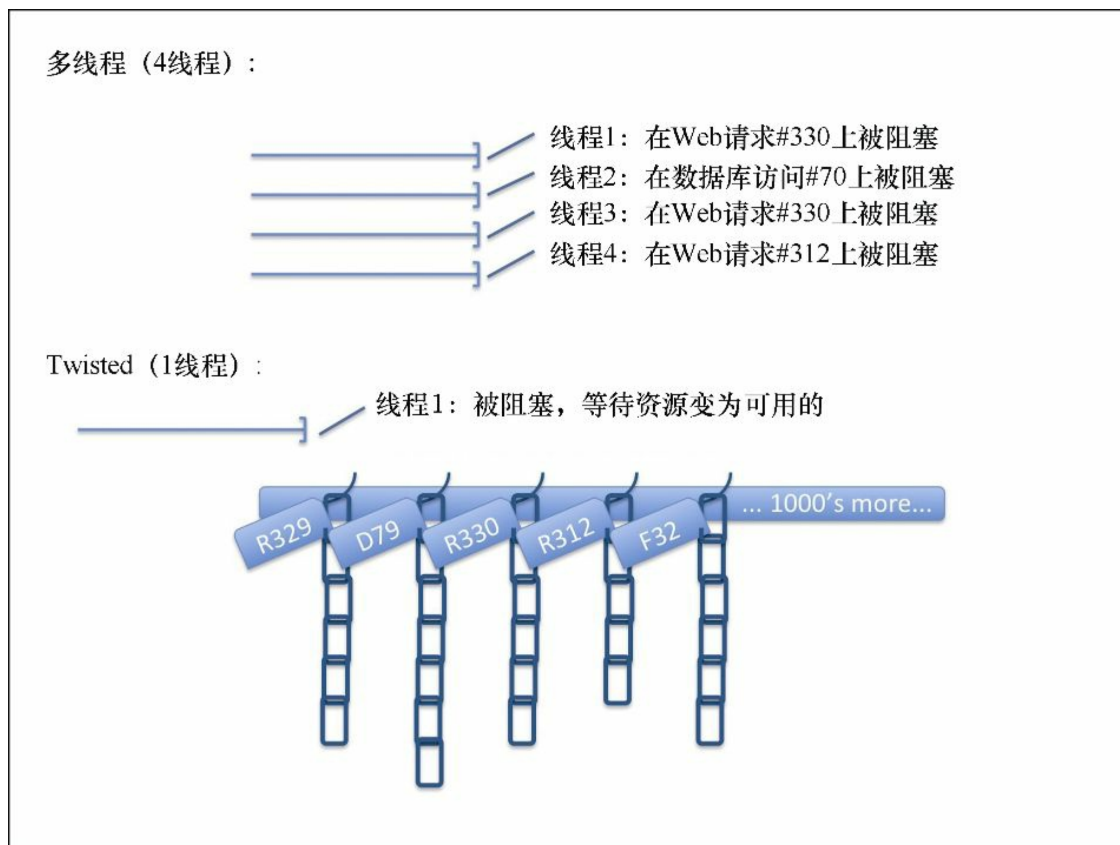


图8.1　多线程代码和Twisted异步代码的对比

Twisted/Scrapy的方式更倾向于尽可能使用单线程。它使用现代操作系统的I/O复用功能（参见`select()`、`poll()`和`epoll()`）作为“挂起器”。在通常会有阻塞操作的地方，比如`result = i_block()`，Twisted提供了一个可以立即返回的替代实现。不过，它并不是返回真实值，而是返回一个hook，比如`deferred = i_dont_block()`，在这

里可以挂起任何想要运行的功能，而不用管什么时候返回值可用（比如，`deferred.addCallback(process_result)`）。一个Twisted应用是由一组此类延迟运行的操作组成的。Twisted唯一的主线程被称为Twisted事件反应器线程，用于监控挂起器，等待某个资源变为可用（比如，服务器返回响应到我们的`Request`中）。当该事件发生时，将会触发链中最前面的延迟操作，执行一些计算，然后依次触发下面的操作。部分延迟操作可能会引发进一步的I/O操作，这样就会造成延迟操作链回到挂起器中，如果可能的话，还会释放CPU以执行其他功能。由于我们使用的是单线程，因此不会存在额外线程所需的上下文切换以及保存资源（如内存）所带来的开销。也就是说，我们使用该非阻塞架构时，只需一个线程，就能达到类似使用数千个线程才能达到的性能。

坦率地说，操作系统开发人员花费了数十年的时间优化线程操作，以使它们速度更快。性能的争论没有以前那么强烈了。有一件大家都认同的事情是，为复杂应用编写正确的线程安全代码非常困难。当你克服考虑延迟/回调所带来的最初冲击后，会发现Twisted代码要比多线程代码简单得多。`inlineCallbacks`生成器工具使得代码更加简单，我们将会在后续章节进一步讨论它。



可以说，到目前为止，最成功的非阻塞I/O系统是Node.js，主要是因为它以高性能和并发性作为出发点，没有人去争论这是好事还是坏事。每个Node.js应用都只用非阻塞API。在Java的世界里，Netty可能是最成功的NIO框架驱动应用，比如Apache Storm和Spark。C++ 11的`std::future`和`std::promise`（与延迟操作非常类似）通过使用libevent或纯POSIX这些库，使得编写异步代码更加简单。

8.1.1　延迟和延迟链

延迟机制是Twisted提供的最基础的机制，能够帮助我们编写异步代码。Twisted API使用延迟机制，允许我们定义发生某些事件时所采取的动作序列。下面让我们具体看一下。



你可以从GitHub上获取本书的全部源代码。如果想要下载本书代码，可以使用git clone `https://github.com/ scalingexcellence/scrapybook`。

本章的完整代码在`ch08`目录中，其中本示例的代码在`ch08/deferreds.py`文件中，你可以使用`./deferreds.py 0`运行该代码。

可以使用Python控制台运行如下的交互式实验。

```
$ python

>>> from twisted.internet import defer

>>> # Experiment 1
```

```
>>> d = defer.Deferred()
>>> d.called
False
>>> d.callback(3)
>>> d.called
True
>>> d.result
3
```

可以看到，Deferred 本质上代表的是一个无法立即获取的值。当触发d 时（调用其callback 方法），其called 状态变为True，而result 属性被设置为在回调方法中设定的值。

```
>>> # Experiment 2

>>> d = defer.Deferred()

>>> def foo(v):

...     print "foo called"

...     return v+1
```

```
...

>>> d.addCallback(foo)

<Deferred at 0x7f...>

>>> d.called

False

>>> d.callback(3)

foo called

>>> d.called
```

```
True

>>> d.result

4
```

延迟机制最强大的功能就是可以在设定值时串联其他要被调用的操作。在上面的例子中，添加了一个foo() 函数作为d 的回调函数。当通过调用callback(3) 触发d 时，会调用函数foo()，打印消息，并将其返回值设为d 最终的result 值。

```
>>> # Experiment 3

>>> def status(*ds):
```

```
...     return [(getattr(d, 'result', "N/A"), len(d.callbacks)) for d in
ds]
>>> def b_callback(arg):
...     print "b_callback called with arg =", arg
...     return b
>>> def on_done(arg):
...     print "on_done called with arg =", arg
...     return arg
```

```
>>> # Experiment 3.a

>>> a = defer.Deferred()

>>> b = defer.Deferred()

>>> a.addCallback(b_callback).addCallback(on_done)

>>> status(a, b)

[('N/A', 2), ('N/A', 0)]

>>> a.callback(3)

b_callback called with arg = 3
```

```
>>> status(a, b)
[(<Deferred at 0x10e7209e0>, 1), ('N/A', 1)]
>>> b.callback(4)
on_done called with arg = 4
>>> status(a, b)
[(4, 0), (None, 0)]
```

该示例展示了更加复杂的延迟行为。我们看到该示例中有一个普通的延迟a，和之前例子中创建的一样，不过这次它有两个回调方法。第一个是b_callback()，返回值是另一个延迟b，而不是一个值。第二个是on_done() 打印函数。我们还有一个status() 函数，用于打印延迟状态。在两个延迟完成初始化之后，得到了相同的状态：[('N/A', 2), ('N/A', 0)]，这意味着两个延迟都还没有被触发，并且第一个延迟有两个回调，而第二个没有回调。然后，当触发第一个延迟时，我们得到了一个奇怪的[(<Deferred at 0x10e7209e0>, 1), ('N/A', 1)] 状态，可以看出现在a 的值是一个延迟（实际上就是b 延迟），并且目前它还有一个回调，这种情况是合理的，因为b_callback() 已经被调用，只剩下了on_done()。意外的情况是现在b 也包含了一个回调。实际上是在后台注册了一个回调，一旦触发b，就会更新它的值。当其发生时，on_done() 依然会被调用，并且最终状态会是[(4, 0), (None, 0)]，和我们预期的一样。

```
>>> # Experiment 3.b

>>> a = defer.Deferred()

>>> b = defer.Deferred()
```

```
>>> a.addCallback(b_callback).addCallback(on_done)

>>> status(a, b)

[('N/A', 2), ('N/A', 0)]

>>> b.callback(4)

>>> status(a, b)

[('N/A', 2), (4, 0)]

>>> a.callback(3)

b_callback called with arg = 3
```

```
on_done called with arg = 4

>>> status(a, b)

[(4, 0), (None, 0)]
```

而另一方面，如果像Experiment3.b 所示，b 先于a 被触发，状态将会变为[('N/A', 2), (4, 0)]，然后当a 被触发时，两个回调都会被调用，最终状态与之前一样。有意思的是，不管顺序如何，最终结果都是相同的。两个例子唯一的不同是，在第一个例子中，b 值保持延迟的时间更长一些，因为它是第二个被触发的，而在第二个例子中，b 首先被触发，并且从该时刻起，它的值就会在需要时被立即使用。

此时，你应该已经对什么是延迟、它们是如何串联起来表示尚不可用的值，有了不错的理解。我们将通过第4个例子结束这一部分的研究，在该示例中，将展示如何触发依赖于多个其他延迟的方法。在Twisted的实现中，将会使用defer.DeferredList 类。

```
>>> # Experiment 4

>>> deferreds = [defer.Deferred() for i in xrange(5)]

>>> join = defer.DeferredList(deferreds)

>>> join.addCallback(on_done)

>>> for i in xrange(4):

...     deferreds[i].callback(i)

>>> deferreds[4].callback(4)

on_done called with arg = [(True, 0), (True, 1), (True, 2),
```

```
                    (True, 3), (True, 4)]
```

可以注意到，尽管for 循环语句只触发了5个延迟中的4个，on_done() 仍然需要等到列表中所有延迟都被触发后才会调用，也就是说，要在最后的deferreds[4].callback() 之后调用。on_done() 的参数是一个元组组成的列表，每个元组对应一个延迟，其中包含两个元素，分别是表示成功的True 或表示失败的False，以及延迟的值。

8.1.2　理解Twisted和非阻塞I/O——一个Python故事

既然我们已经掌握了原语，接下来让我告诉你一个Python的小故事。该故事中所有人物均为虚构，如有雷同纯属巧合。

```
# ~*~ Twisted - A Python tale ~*~

from time import sleep

# Hello, I'm a developer and I mainly setup Wordpress.
def install_wordpress(customer):
    # Our hosting company Threads Ltd. is bad. I start installation and...
    print "Start installation for", customer
    # ...then wait till the installation finishes successfully. It is
    # boring and I'm spending most of my time waiting while consuming
    # resources (memory and some CPU cycles). It's because the process
    # is *blocking*.
    sleep(3)
```

```
    print "All done for", customer

# I do this all day long for our customers
def developer_day(customers):
    for customer in customers:
        install_wordpress(customer)

developer_day(["Bill", "Elon", "Steve", "Mark"])
```

运行该代码。

```
$ ./deferreds.py 1
------ Running example 1 ------
Start installation for Bill
All done for Bill
Start installation
...
* Elapsed time: 12.03 seconds
```

我们得到的是顺序的执行。4位客户，每人执行3秒，意味着总共需要12秒的时间。这种方式的扩展性不是很好，因此我们将在第二个例子中添加多线程。

```
import threading

# The company grew. We now have many customers and I can't handle
the
# workload. We are now 5 developers doing exactly the same thing.
def developers_day(customers):
    # But we now have to synchronize... a.k.a. bureaucracy
    lock = threading.Lock()
    #
    def dev_day(id):
        print "Goodmorning from developer", id
```

```
        # Yuck - I hate locks...
        lock.acquire()
        while customers:
            customer = customers.pop(0)
            lock.release()
            # My Python is less readable
            install_wordpress(customer)
            lock.acquire()
        lock.release()
        print "Bye from developer", id
    # We go to work in the morning
    devs = [threading.Thread(target=dev_day, args=(i,)) for i in
range(5)]
    [dev.start() for dev in devs]
    # We leave for the evening
    [dev.join() for dev in devs]

# We now get more done in the same time but our dev process got more
# complex. As we grew we spend more time managing queues than doing dev
# work. We even had occasional deadlocks when processes got extremely
# complex. The fact is that we are still mostly pressing buttons and
# waiting but now we also spend some time in meetings.
developers_day(["Customer %d" % i for i in xrange(15)])
```

按照下述方式运行这段代码。

```
$ ./deferreds.py 2

------ Running example 2 ------

Goodmorning from developer 0Goodmorning from developer
```

```
1Start installation forGoodmorning from developer 2

Goodmorning from developer 3Customer 0

...

from developerCustomer 13 3Bye from developer 2

* Elapsed time: 9.02 seconds
```

在这段代码中，使用了5个线程并行执行。15个客户，每人3秒，总共需要执行45秒，而当使用5个并行的线程时，最终只花费了9秒钟。不过代码有些难看。现在代码的一部分只用于管理并发性，而不是专注于算法或业务逻辑。另外，输出也变得混乱并且可读性很差。即使是让很

简单的多线程代码正确运行，也有很大难度，因此我们将转为使用 Twisted。

```
# For years we thought this was all there was... We kept hiring more
# developers, more managers and buying servers. We were trying harder
# optimising processes and fire-fighting while getting mediocre
# performance in return. Till luckily one day our hosting
# company decided to increase their fees and we decided to
# switch to Twisted Ltd.!
from twisted.internet import reactor
from twisted.internet import defer
from twisted.internet import task

# Twisted has a slightly different approach
def schedule_install(customer):
    # They are calling us back when a Wordpress installation completes.
    # They connected the caller recognition system with our CRM and
    # we know exactly what a call is about and what has to be done
    # next.
    #
    # We now design processes of what has to happen on certain events.
    def schedule_install_wordpress():
        def on_done():
            print "Callback: Finished installation for", customer
        print "Scheduling: Installation for", customer
        return task.deferLater(reactor, 3, on_done)
    #
    def all_done(_):
        print "All done for", customer
    #
    # For each customer, we schedule these processes on the CRM
    # and that
    # is all our chief-Twisted developer has to do
    d = schedule_install_wordpress()
    d.addCallback(all_done)
    #
    return d

# Yes, we don't need many developers anymore or any synchronization.
# ~~ Super-powered Twisted developer ~~
def twisted_developer_day(customers):
    print "Goodmorning from Twisted developer"
    #
    # Here's what has to be done today
    work = [schedule_install(customer) for customer in customers]
```

```
    # Turn off the lights when done
    join = defer.DeferredList(work)
    join.addCallback(lambda _: reactor.stop())
    #
    print "Bye from Twisted developer!"
# Even his day is particularly short!
twisted_developer_day(["Customer %d" % i for i in xrange(15)])

# Reactor, our secretary uses the CRM and follows-up on events!
reactor.run()
```

现在运行该代码。

```
$ ./deferreds.py 3

------ Running example 3 ------

Goodmorning from Twisted developer

Scheduling: Installation for Customer 0

....
```

```
Scheduling: Installation for Customer 14

Bye from Twisted developer!

Callback: Finished installation for Customer 0

All done for Customer 0

Callback: Finished installation for Customer 1

All done for Customer 1

...

All done for Customer 14
```

```
* Elapsed time: 3.18 seconds
```

此时，我们在没有使用多线程的情况下，就获得了良好运行的代码，以及漂亮的输出结果。我们并行处理了所有的15位客户，也就是说，应当执行45秒的计算只花费了3秒钟！技巧就是将所有阻塞调用的`sleep()`替换为Twisted对应的`task.deferLater()`以及回调函数。由于处理现在发生在其他地方，因此可以毫不费力地同时为15位客户服务。



刚才提到前面的处理此时是在其他地方执行的。这是在作弊吗？答案当然不是。算法计算仍然在CPU中处理，不过与磁盘和网络操作相比，CPU操作速度很快。因此，将数据传给CPU、从一个CPU发送或存储数据到另一个CPU中，占据了大部分时间。我们使用非阻塞的I/O操作，为CPU节省了这些时间。这些操作，尤其是像`task.deferLater()`这样的操作，会在数据传输完成后触发回调函数。

另一个需要非常注意的地方是`Goodmorning from Twisted developer`以及`Bye from Twisted developer!`消息。在代码启动

时，它们就都被立即打印了出来。如果代码过早地到达该点，那么应用实际是什么时候运行的呢？答案是Twisted应用（包括Scrapy）完全运行在`reactor.run()`上！当调用该方法时，必须拥有应用程序预期使用的所有可能的延迟链（相当于前面故事中建立CRM系统的步骤和流程）。你的`reactor.run()`（故事中的秘书）执行事件监控以及触发回调。



reactor的主要规则是：只要是快速的非阻塞操作就可以做任何事。

非常好！不过虽然代码没有了多线程时的混乱输出，但是这里的回调函数还是有一些难看！因此，我们将引入下一个例子。

```
# Twisted gave us utilities that make our code way more readable!
@defer.inlineCallbacks
def inline_install(customer):
    print "Scheduling: Installation for", customer
    yield task.deferLater(reactor, 3, lambda: None)
    print "Callback: Finished installation for", customer
    print "All done for", customer

def twisted_developer_day(customers):
   ... same as previously but using inline_install()
       instead of schedule_install()

twisted_developer_day(["Customer %d" % i for i in xrange(15)])
reactor.run()
```

以如下方式运行该代码。

```
$ ./deferreds.py 4

... exactly the same as before
```

上述代码和之前那个版本的代码看起来基本一样，不过更加优雅。inlineCallbacks 生成器使用了一些Python机制让 inline_install() 的代码能够暂停和恢复。inline_install() 变为延迟函数，并且为每位客户并行执行。每当执行yield 时，执行会在当前的inline_install() 实例上暂停，当yield的延迟函数触发时再恢复。

现在唯一的问题是，如果不是只有15个客户，而是10000个，该代码会无耻地同时启动10000个处理序列（调用HTTP请求、数据库写操作等）。这样可能会正常运行，也可能造成各种各样的失败。在大规模并发应用中，比如Scrapy，一般需要将并发量限制到可接受的水平。在本例中，可以使用task.Cooperator() 实现该限制。Scrapy使用了同样的机制在item处理管道中限制并发量（CONCURRENT_ITEMS 设置）。

```
@defer.inlineCallbacks
def inline_install(customer):
```

```
    ... same as above

# The new "problem" is that we have to manage all this concurrency to
# avoid causing problems to others, but this is a nice problem to have.
def twisted_developer_day(customers):
    print "Goodmorning from Twisted developer"
    work = (inline_install(customer) for customer in customers)
    #
    # We use the Cooperator mechanism to make the secretary not
    # service more than 5 customers simultaneously.
    coop = task.Cooperator()
    join = defer.DeferredList([coop.coiterate(work) for i in xrange(5)])
    #
    join.addCallback(lambda _: reactor.stop())
    print "Bye from Twisted developer!"

twisted_developer_day(["Customer %d" % i for i in xrange(15)])
reactor.run()

# We are now more lean than ever, our customers happy, our hosting
# bills ridiculously low and our performance stellar.

# ~*~ THE END ~*~
```

运行该代码。

```
$ ./deferreds.py 5

------ Running example 5 ------

Goodmorning from Twisted developer
```

```
Bye from Twisted developer!

Scheduling: Installation for Customer 0

...

Callback: Finished installation for Customer 4

All done for Customer 4

Scheduling: Installation for Customer 5

...

Callback: Finished installation for Customer 14
```

```
All done for Customer 14

* Elapsed time: 9.19 seconds
```

可以看到，现在有类似于5个客户的处理槽。如果想要处理一个新的客户，只有在存在空槽时才可以开始，实际上，在这个例子中客户处理的时间总是相同的（3秒），因此会造成5位客户会在同一时间被批量处理的情况。最后，我们得到了和多线程示例中相同的性能，不过现在只使用了一个线程，代码更加简单并且更容易正确编写。

祝贺你，坦白地说，现在你得到了对于Twisted和非阻塞I/O编程的一份非常严谨的介绍。

8.2　Scrapy架构概述

图8.2所示为Scrapy的架构。

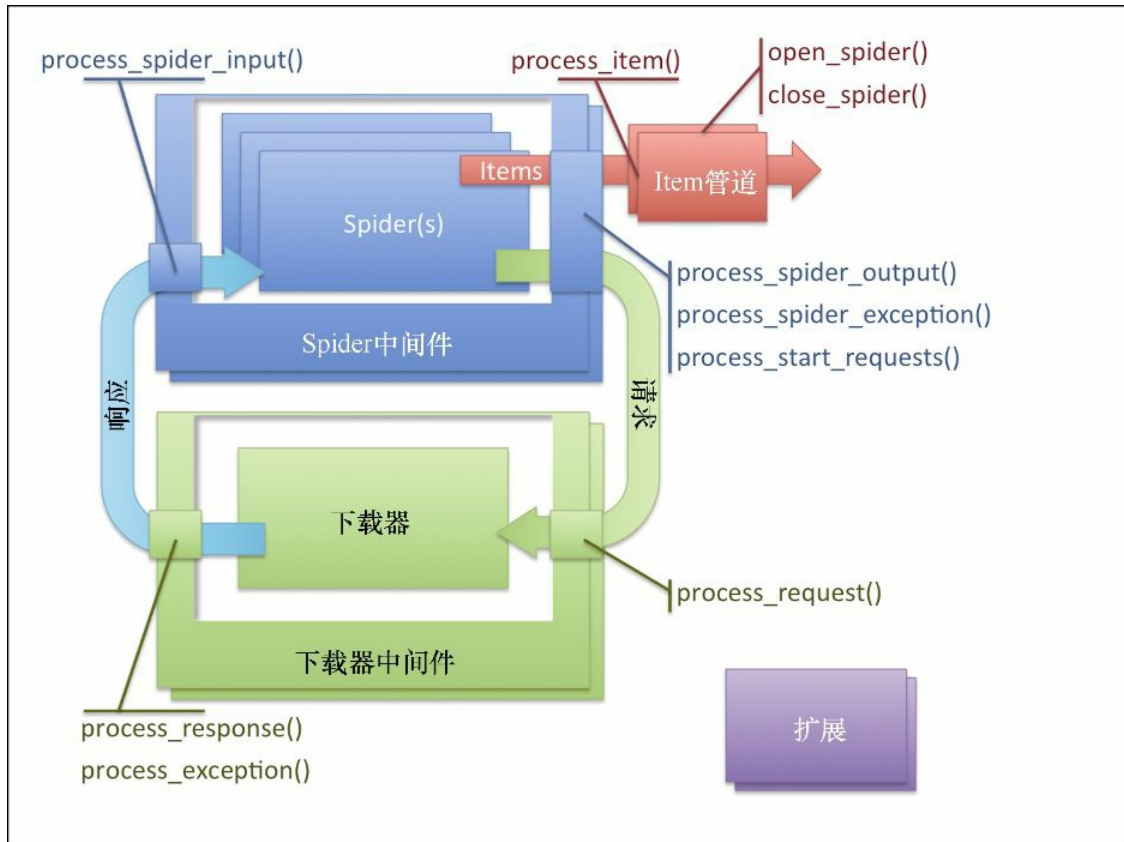


图8.2　Scrapy架构

你可能已经注意到，该架构运行在我们熟悉的三类对象之上：`Request` 、`Response` 以及`Item` 。我们的爬虫就在架构的核心位置，它们创建`Request` ，处理`Response` ，生成`Item` 和更多的`Request` 。

爬虫生成的每个`Item` 都使用其`process_item()` 方法由Item管道序列执行后置处理。通常情况下，`process_item()` 会修改`Item` ，然后以返回这些`Item` 的方式将其传给后续的管道。有时候（比如冗余或非法数据的情况），我们可能需要放弃一个Item，此时可以通过抛出`DropItem` 异常的方式实现。这种情况下，后续的管道将不会再接收该Item。如果我们提供了`open_spider()` 和/ 或`close_spider()` 方法，

那么爬虫会对应地在开始和结束爬虫时调用该方法。这里是我们进行初始化和清理工作的时机。Item管道通常用于执行问题域名或基础结构的操作，比如清理数据、向数据库插入Item 等。你还会发现自己会在项目之间很大程度地复用它们，尤其是当处理基础架构细节时。第4章中使用过的Appery.io管道，即通过少量配置上传Item 到Appery.io的工作，就是用Item管道执行基础架构工作的一个例子。

我们通常会从爬虫发送Request ，并得到返回的Response ，来进行工作。Scrapy以透明的方式负责Cookie、权限认证、缓存等，我们所需要做的就是偶尔调整一些设置。这其中大部分功能是在下载器中间件中实现的。它们通常都非常复杂，在处理Request/Response 内部构件时有着很高的技巧。你可以创建自定义的中间件，以使Scrapy按照你要求的方式处理Request 。通常，成功的中间件可以在多个项目中复用，并且可以向其他Scrapy开发者提供有用的功能，因此向社区分享是个不错的选择。你没有必要经常编写下载器中间件。如果你想了解默认的下载器中间件，可以查看Scrapy的Github仓库中settings/default_settings.py 文件的DOWNLOADER_MIDDLEWARES_BASE 设置。

下载器是真正执行下载的引擎。除非你是Scrapy的代码贡献者，否则不要修改它。

有时候，你可能需要编写爬虫中间件（见图8.3）。这些中间件在爬虫之后且所有下载器中间件之前处理Request ；而在处理Response 时，则是相反的顺序。使用下载器中间件，可以做很多事情，比如重写所有URL，使用HTTPS代替HTTP，而不用管爬虫从页面中抽取出来的

内容是什么。它可以实现特定于项目需求的功能，并分享给所有的爬虫。下载器中间件和爬虫中间件最主要的区别是，当下载器中间件获取一个`Request`时，只会返回一个`Response`。而爬虫中间件可以在对某些`Request`不感兴趣时舍弃掉它们，或者对每个输入的`Request`都发出多个`Request`，用来完成你的应用程序的目标。可以说爬虫中间件是针对`Request`和`Response`的，而Item管道是针对`Item`的。爬虫中间件同样也接收`Item`，不过通常情况下不会对其进行修改，因为在Item管道中进行这些操作更加容易。如果你想了解默认的爬虫中间件，可以在Scrapy的Git上查看`settings/default_settings.py`文件的`SPIDER_MIDDLEWARES_BASE`设置。

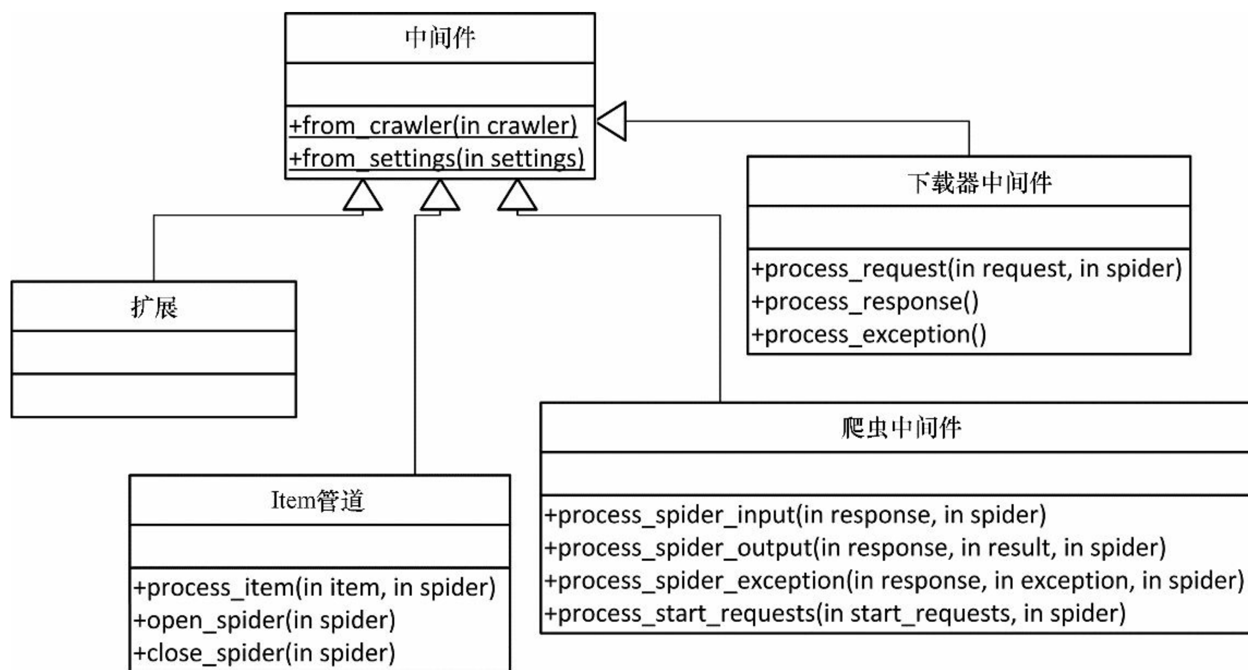


图8.3　中间件架构

最后还有一个部分是扩展。扩展非常常见，实际上其常见程度仅次于Item管道。它们是在爬取工作启动时加载的普通类，可以访问设置、

爬虫、注册回调信号以及定义自己的信号。信号是一类基础的Scrapy API，它可以让回调函数在系统中发生某些事情时进行调用，比如Item被抓取、丢弃时或爬虫开启时。有很多非常有用的预定义信号，我们将会在后边见到其中的一部分。某种意义上讲，扩展有些博而不精，它能够让你写出任何可以想到的工具，但又无法给你实际的帮助（比如像Item管道的`process_item()`方法）。我们必须将其hook到信号上，自己实现需要的功能。例如，在达到指定页数或Item个数后停止爬取，就是通过扩展实现的。如果想要了解默认的扩展，可以从Scrapy的Git上查看`settings/default_settings.py`文件的`EXTENSIONS_BASE`设置。

更严格地说，Scrapy把所有这些类都当作是中间件（通过`MiddlewareManager`类的子类管理），允许我们通过实现`from_crawler()`或`from_settings()`类方法，分别从`Crawler`或`Settings`对象初始化它们。由于`Settings`可以从`Crawler`中轻松获取（`crawler.settings`），因此`from_crawler()`是更加流行的方式。如果不需要`Settings`或`Crawler`，可以不去实现它们。

表8.1可以帮助你在针对指定问题时决定最好的机制。

表**8.1**

问题	解决方案
一些只针对于我正在爬取的网站的内容	修改爬虫
修改或存储Item——特定领域，可能在项目间复用	编写Item管道

修改或丢弃Request/Response ——特定领域，可能在项目间复用	编写爬虫中间件
执行Request/Response ——通用，比如支持一些定制化登录模式或处理Cookie的特定方式	编写下载器中间件
所有其他问题	编写扩展

8.3　示例1：非常简单的管道

假设我们有一个包含几个爬虫的应用，以Python常见格式提供爬取日期。数据库需要将其转为字符串格式，以便进行索引。我们不想编辑爬虫，因为爬虫的数量比较多。此时可以怎么做呢？使用一个非常简单的管道对Item进行后置处理，执行需要的转换即可。让我们看看它是如何工作的。

```
from datetime import datetime

class TidyUp(object):
    def process_item(self, item, spider):
        item['date'] = map(datetime.isoformat, item['date'])
        return item
```

如你所见，这里只有一个包含`process_item()`方法的简单类。这是我们为了这个简单管道所需要做的所有事情。我们可以复用第3章中的爬虫，将前面的代码写入`pipelines`目录的`tidyup.py`文件中。



可以将这个Item管道的代码放到任何地方，不过为其创建一个单独的目录是一个好主意。

现在，需要编辑项目的settings.py 文件，将ITEM_PIPELINES 设置为

```
ITEM_PIPELINES = {'properties.pipelines.tidyup.TidyUp': 100 }
```

前面代码字典中的数字100，用于定义管道连接的顺序。如果另一个管道有更小的数值，它将在该管道之前优先处理Item 。



本示例的完整代码位于ch08/properties 目录中。

现在，可以运行该爬虫了。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90

...
```

```
INFO: Enabled item pipelines: TidyUp
...
DEBUG: Scraped from <200 ...property_000060.html>
...
   'date': ['2015-11-08T14:47:04.148968'],
```

和我们期望的一样，日期现在被格式化为ISO字符串了。

8.4　信号

信号提供了一种为系统中发生的事件添加回调的机制，比如当爬虫开启时或当item被抓取时。可以使用crawler.signals.connect()方法hook到它们上（下一节将会给出使用它的一个示例）。信号总共有11个，理解它们的最简单方式可能就是在实践中看到它们。我创建了一个项目，在其中创建了一个扩展，hook了所有可以使用的信号。另外，我还创建了一个Item管道、一个下载器和一个爬虫中间件，可以记录所有的方法调用。该项目使用的爬虫非常简单，只对两个item进行yield操作，然后抛出异常。

```
def parse(self, response):
    for i in range(2):
        item = HooksasyncItem()
        item['name'] = "Hello %d" % i
        yield item
    raise Exception("dead")
```

在第二个item中，我配置了Item管道，以抛出DropItem异常。



本示例的完整代码可以从ch08/hooksasync得到。

使用该项目，我们可以更好地理解某个信号是什么时候发出的。请查看如下执行中日志行之间的注释（为了简短起见，省略了部分行）。

```
$ scrapy crawl test
```

```
... many lines ...

# First we get those two signals...

INFO: Extension, signals.spider_opened fired

INFO: Extension, signals.engine_started fired

# Then for each URL we get a request_scheduled signal

INFO: Extension, signals.request_scheduled fired

...# when download completes we get response_downloaded

INFO: Extension, signals.response_downloaded fired
```

```
INFO: DownloaderMiddlewareprocess_response called for example.com

# Work between response_downloaded and response_received

INFO: Extension, signals.response_received fired

INFO: SpiderMiddlewareprocess_spider_input called for example.com

# here our parse() method gets called... and then SpiderMiddleware used

INFO: SpiderMiddlewareprocess_spider_output called for example.com

# For every Item that goes through pipelines successfully...

INFO: Extension, signals.item_scraped fired
```

```
# For every Item that gets dropped using the DropItem exception...

INFO: Extension, signals.item_dropped fired

# If your spider throws something else...

INFO: Extension, signals.spider_error fired

# ... the above process repeats for each URL

# ... till we run out of them. then...

INFO: Extension, signals.spider_idle fired

# by hooking spider_idle you can schedule further Requests. If you don't
```

```
# the spider closes.

INFO: Closing spider (finished)

INFO: Extension, signals.spider_closed fired

# ... stats get printed

# and finally engine gets stopped.

INFO: Extension, signals.engine_stopped fired
```

虽然只有11个信号，可能会感觉比较有限，但是每个Scrapy的默认中间件都是只使用它们实现的，因此它们肯定够用。请注意，除了spider_idle 、spider_error 、request_scheduled 、response_received 和response_downloaded 以外的所有其他信号，都可以返回延迟，而不是真实值。

8.5　示例2：测量吞吐量和延时的扩展

当我们在第9章中添加管道后，测量吞吐量（每秒的item数）和延时（计划后和下载后的时间）的变化是一件很有意思的事情。

Scrapy扩展中已经包含了一个测量吞吐量的扩展，即日志统计扩展（scrapy/extensions/logstats.py ），我们将会以此为起点。要想测量延时，需要hook一些信号，包括request_scheduled 、response_received 和item_scraped 。我们对每个信号记录时间戳，并通过累计多次取平均值的方式减去适当的计算延时。通过观察这些信号提供的回调参数，会发现一些讨厌的东西。item_scraped 只在Response 中获得，request_scheduled 只在Request 中获得，而response_received 则是两者中都有。幸运的是，我们不需要任何特殊的技巧来传递这些值。每个Response 都有一个Request 成员，回指其Request ，更好的是它拥有我们在第5章中看到的meta 字典，它和原始Request 中的一样，而不管是否存在重定向。非常好，我们可以在这里存储时间戳了！



实际上，这并不是我的主意。同样的机制已经在AutoThrottle扩展（scrapy/extensions/throttle.py ）中使用了。在该扩展中，使用了request.meta.get('download_latency')，其中download_latency 是在scrapy/core/downloader/webclient.py 下载器中进行计算的。在编写中间件时，最快的改善方式就是让自己熟悉Scrapy默认的中间件代码。

下面是扩展的代码。

```
class Latencies(object):
   @classmethod
   def from_crawler(cls, crawler):
     return cls(crawler)

   def __init__(self, crawler):
     self.crawler = crawler
     self.interval = crawler.settings.getfloat('LATENCIES_INTERVAL')
     if not self.interval:
        raise NotConfigured
     cs = crawler.signals
     cs.connect(self._spider_opened, signal=signals.spider_opened)
     cs.connect(self._spider_closed, signal=signals.spider_closed)
     cs.connect(self._request_scheduled, signal=signals.request_scheduled)
     cs.connect(self._response_received, signal=signals.response_received)
     cs.connect(self._item_scraped, signal=signals.item_scraped)
     self.latency, self.proc_latency, self.items = 0, 0, 0

   def _spider_opened(self, spider):
     self.task = task.LoopingCall(self._log, spider)
     self.task.start(self.interval)

   def _spider_closed(self, spider, reason):
     if self.task.running:
         self.task.stop()

   def _request_scheduled(self, request, spider):
     request.meta['schedule_time'] = time()
   def _response_received(self, response, request, spider):
     request.meta['received_time'] = time()
```

```
  def _item_scraped(self, item, response, spider):
    self.latency += time() - response.meta['schedule_time']
    self.proc_latency += time() - response.meta['received_time']
    self.items += 1
  def _log(self, spider):
    irate = float(self.items) / self.interval
    latency = self.latency / self.items if self.items else 0
    proc_latency = self.proc_latency / self.items if self.items else 0
    spider.logger.info(("Scraped %d items at %.1f items/s, avg latency: "
      "%.2f s and avg time in pipelines: %.2f s") %
      (self.items, irate, latency, proc_latency))
    self.latency, self.proc_latency, self.items = 0, 0, 0
```

前两个方法非常重要，因为它们很通用。它们使用`Crawler`对象初始化中间件。你会发现这些代码几乎出现在每个重要的中间件当中。`from_crawler(cls, crawler)`是获取`Crawler`对象的方式。然后，可以注意到在`__init__()`方法中，访问了`crawler.settings`，并且会在其未设置时抛出`NotConfigured`异常。你会看到很多`FooBar`扩展，用于检查相应的`FOOBAR_ENABLED`设置，如果没有设置或者设置为`False`时，将会抛出异常。这是一种非常常见的模式，是为了方便将中间件包含在对应的`settings.py`设置中（比如`ITEM_PIPELINES`），但是默认情况下是禁用的，除非通过其对应的设置显式启用。许多默认的Scrapy中间件（比如AutoThrottle或HttpCache）都使用了这种模式。在本例中，我们的扩展会保持`LATENCIES_INTERVAL`的禁用状态，除非已经对其进行了设置。

在`__init__()`方法的后面一部分代码中，我们使用`crawler.signals.connect()`，为所有感兴趣的信号都注册了回调，并初始化了一些成员变量。这个类的剩余部分实现了信号处理器。

在_spider_opened()中，我们初始化了一个计时器，会每隔LATENCIES_INTERVAL秒调用_log()方法；在_spider_closed()中，我们停止了该计时器。在_request_scheduled()和_response_received()中，我们在request.meta中存储了时间戳；而在_item_scraped()中，我们累计两次延时（从计划/接收开始直到当前时间），并递增抓取到的Item的数量。在_log()方法中，我们计算了平均值，格式化并打印出消息，重置累加器以开始另一个采样周期。



任何在多线程上下文中编写类似代码的人，都会意识到上述代码中没有使用互斥锁。本例可能还不是特别复杂，不过编写单线程代码仍然要更加简单，并且在更加复杂的场景下可以很好地扩展。

我们可以将该扩展的代码添加到latencies.py模块中，放到和settings.py同级的目录下。如果想要启用该扩展，只需在settings.py文件中添加如下两行。

```
EXTENSIONS = { 'properties.latencies.Latencies': 500, }
LATENCIES_INTERVAL = 5
```

我们可以像平时那样运行它。

```
$ pwd

/root/book/ch08/properties

$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000 -s LOG_LEVEL=INFO

...

INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)

INFO: Scraped 0 items at 0.0 items/sec, average latency: 0.00 sec and

average time in pipelines: 0.00 sec

INFO: Scraped 115 items at 23.0 items/s, avg latency: 0.84 s and avg time
```

```
in pipelines: 0.12 s

INFO: Scraped 125 items at 25.0 items/s, avg latency: 0.78 s and avg time

in pipelines: 0.12 s
```

日志的第一行来自日志统计扩展，而接下来的各行来自我们的扩展。可以看到吞吐量是每秒25个item，平均时延是0.78秒，我们在下载后几乎没有花费时间处理。通过利特尔法则，我们得到系统中item的数量为N = S · T = 43 · 0.45 ≌ 19。无论设置的CONCURRENT_REQUESTS和CONCURRENT_REQUESTS_PER_DOMAIN是多少，即便没有触及100%的CPU，出于某些原因，也不应该使其超过30。我们可以在第10章中了解更多相关内容。

8.6 中间件延伸

本节是为好奇的读者提供的，而不再是开发者。如果只是编写基础或中级的Scrapy扩展的话，你并不需要了解这些内容。

如果查看`scrapy/settings/default_settings.py`文件，就会发现在默认设置中有很多类名。Scrapy大量使用了依赖注入机制，可以让我们自定义和扩展许多内部对象。例如，一些人可能希望支持除了文件、HTTP、HTTPS、S3以及FTP这些在DOWNLOAD_HANDLERS_BASE设置中定义好的协议以外的更多协议。要想实现这一点，只需要创建一个下载处理器类，并在DOWNLOAD_HANDLERS设置中添加映射即可。最困难的部分是找出你的自定义类必须包含哪些接口（即需要实现哪些方法），因为大部分接口都不是显式的。你必须阅读源代码，查看这些类是如何使用的。最好的办法是从已有的实现开始，将其修改为令自己满意的版本。不过，这些接口在近期的Scrapy版本中已经逐渐趋于稳定，因此我将尝试在图8.4中将它们和Scrapy核心类一起记录成文档（这里省略了前面已经提及的中间件架构）。

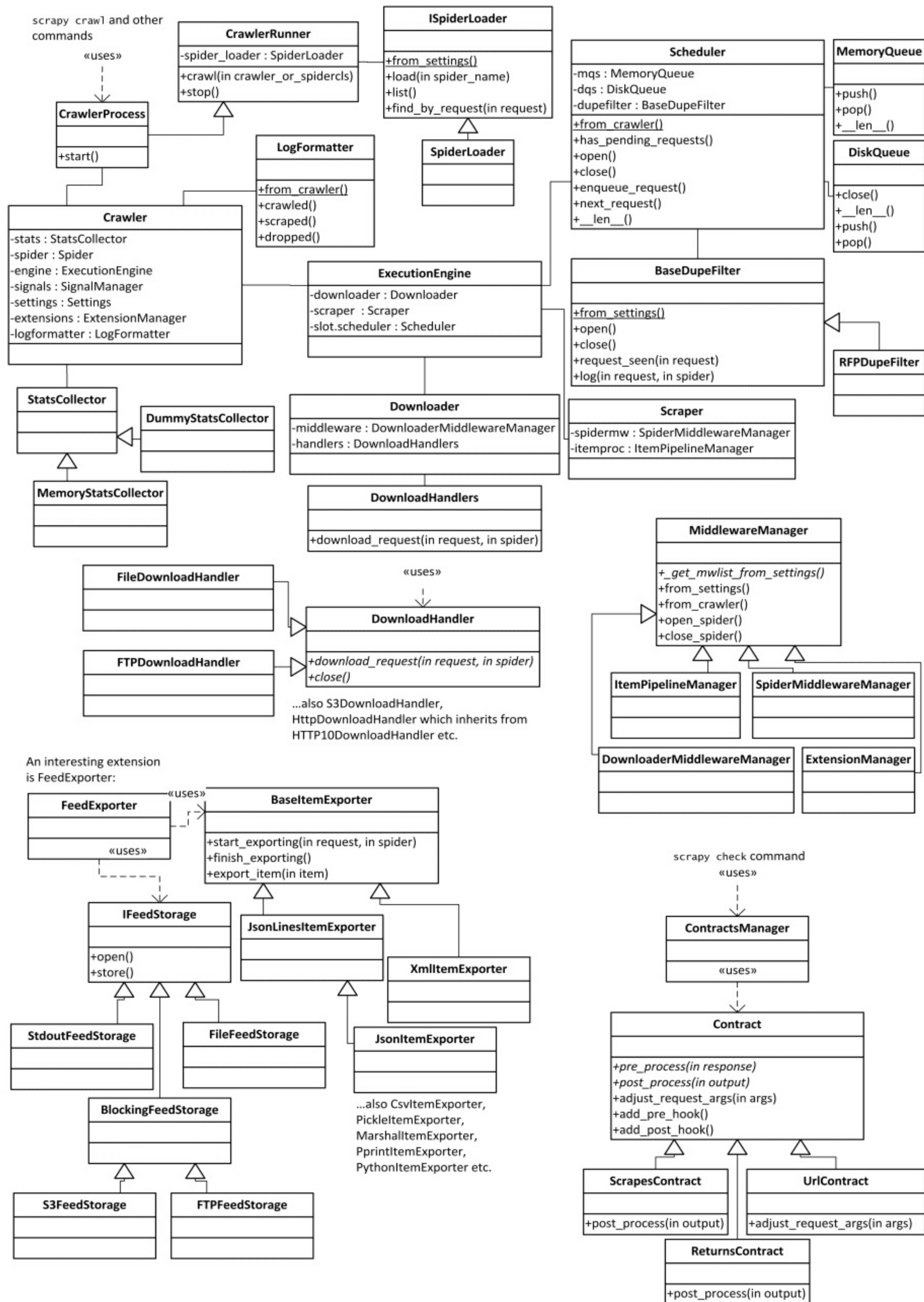

scrapy crawl and other commands
«uses»
CrawlerProcess
+start()
CrawlerRunner
-spider_loader : SpiderLoader
+crawl(in crawler_or_spidercls)
+stop()
ISpiderLoader
+from_settings()
+load(in spider_name)
+list()
+find_by_request(in request)
SpiderLoader
Scheduler
-mqs : MemoryQueue
-dqs : DiskQueue
-dupefilter : BaseDupeFilter
+from_crawler()
+has_pending_requests()
+open()
+close()
+enqueue_request()
+next_request()
+__len__()
MemoryQueue
+push()
+pop()
+__len__()
DiskQueue
+close()
+__len__()
+push()
+pop()
LogFormatter
+from_crawler()
+crawled()
+scraped()
+dropped()
Crawler
-stats : StatsCollector
-spider : Spider
-engine : ExecutionEngine
-signals : SignalManager
-settings : Settings
-extensions : ExtensionManager
-logformatter : LogFormatter
ExecutionEngine
-downloader : Downloader
-scraper : Scraper
-slot.scheduler : Scheduler
BaseDupeFilter
+from_settings()
+open()
+close()
+request_seen(in request)
+log(in request, in spider)
RFPDupeFilter
StatsCollector
DummyStatsCollector
MemoryStatsCollector
Downloader
-middleware : DownloaderMiddlewareManager
-handlers : DownloadHandlers
Scraper
-spidermw : SpiderMiddlewareManager
-itemproc : ItemPipelineManager
DownloadHandlers
+download_request(in request, in spider)
MiddlewareManager
+_get_mwlist_from_settings()
+from_settings()
+from_crawler()
+open_spider()
+close_spider()
FileDownloadHandler
«uses»
DownloadHandler
+download_request(in request, in spider)
+close()
FTPDownloadHandler
...also S3DownloadHandler, HttpDownloadHandler which inherits from HTTP10DownloadHandler etc.
ItemPipelineManager
SpiderMiddlewareManager
DownloaderMiddlewareManager
ExtensionManager
An interesting extension is FeedExporter:
FeedExporter
«uses»
BaseItemExporter
+start_exporting(in request, in spider)
+finish_exporting()
+export_item(in item)
«uses»
IFeedStorage
+open()
+store()
JsonLinesItemExporter
XmlItemExporter
JsonItemExporter
StdoutFeedStorage
FileFeedStorage
BlockingFeedStorage
S3FeedStorage
FTPFeedStorage
...also CsvItemExporter, PickleItemExporter, MarshalItemExporter, PprintItemExporter, PythonItemExporter etc.
scrapy check command
«uses»
ContractsManager
«uses»
Contract
+pre_process(in response)
+post_process(in output)
+adjust_request_args(in args)
+add_pre_hook()
+add_post_hook()
ScrapesContract
+post_process(in output)
UrlContract
+adjust_request_args(in args)
ReturnsContract
+post_process(in output)

图8.4　Scrapy接口和核心对象

核心类位于图8.4的左上角。当人们使用`scrapy crawl`时，Scrapy就会使用`CrawlerProcess`对象创建我们熟悉的`Crawler`对象。`Crawler`对象是最重要的Scrapy类。它包括`settings`、`signals`以及`spider`。在名为`extensions.crawler.engine`的`ExtensionManager`对象中，还包含所有的扩展，这将带领我们来到另一个非常重要的类——`ExecutionEngine`。在该类中，包含了`Scheduler`、`Downloader`以及`Scraper`。URL通过`Scheduler`进行计划，通过`Downloader`下载，通过`Scraper`进行后置处理。毫无疑问，`Downloader`包含`DownloaderMiddleware`和`DownloadHandler`，而`Scraper`包含`SpiderMiddleware`和`ItemPipeline`。4个`MiddlewareManager`也都拥有其自己的小架构。在Scrapy中，feed输出是以扩展的形式实现的，即`FeedExporter`。它包含两个独立的结构，一个用于定义输出格式，而另一个用于存储类型。这就允许我们可以通过调整输出的URL将S3的XML文件导出为命令行上的Pickle编码输出。这两个结构还可以使用`FEED_STORAGES`和`FEED_EXPORTERS`设置进行独立扩展。最后，`scrapy check`命令使用的contract也有其自身的结构，可以使用`SPIDER_CONTRACTS`设置进行扩展。

8.7　本章小结

恭喜你，你已经对`Scrapy`和`Twisted`编程有了深入了解。你可能还会多次阅读本章，并将本章作为参考使用。到目前为止，我们需要的最流行的扩展是`Item`处理管道。下一章会用它解决一些常见的问题。

第9章　管道秘诀

上一章讨论了使用Scrapy中间件的编程技术。本章将通过展示各种常见用例（包括消费REST API、数据库接口、处理CPU密集型任务以及与遗留服务的接口），重点关注编写正确而高效的管道。

在本章中，我们将会使用几个新的服务器，你可以在图9.1的右侧看到这些服务器。

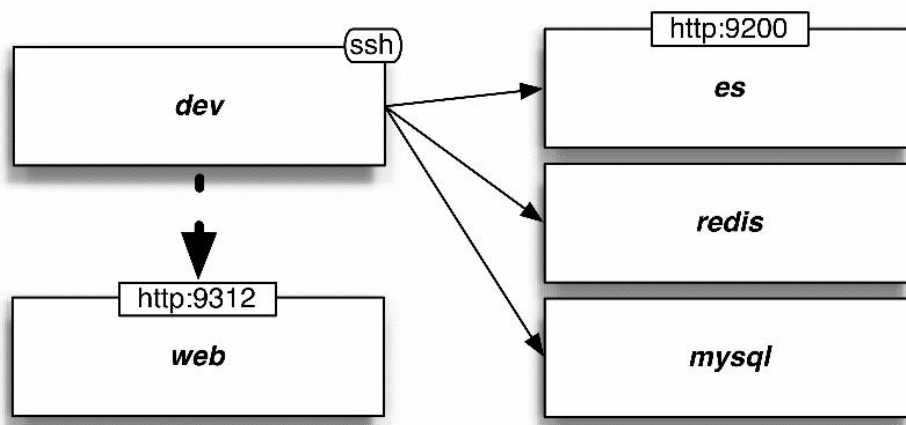


图9.1　本章使用的系统

Vagrant应该已经为我们创建好了这些服务器，我们可以从dev服务器中使用其主机名进行ping操作，例如`ping es`或`ping mysql`。话不多说，让我们从REST API开始探索吧。

9.1　使用REST API

REST是一套用于创建现代Web服务的技术，其主要优点是比SOAP或专有Web服务机制更加简单，更加轻量级。软件开发人员观察发现，Web服务经常提供的**CRUD**（创建、读取、更新、删除 **[Create、Read、Update、Delete]**）功能与HTTP基本操作（GET、POST、PUT、DELETE）具有相似性。另外，他们还发现典型的Web服务调用其所需的大部分信息时，都可以将其压缩到资源URL上。例如，`http://api.mysite.com/ customer/john`是一个资源URL，它可以让我们确定目标服务器（`api.mysite.com`），实际上我正在尝试在服务器上执行和`customers`（表）相关的操作，更具体的说就是执行和`john`（行——主键）相关的操作。当它与其他Web概念（如安全认证、无状态、缓存、使用XML或JSON作为载荷等）结合时，能够通过一种强大而又简单、熟悉且可以轻松跨平台的方式，提供和使用Web服务。难怪REST可以掀起软件行业的一场风暴。

9.1.1 使用treq

`treq`是一个Python包，相当于基于Twisted应用编写的Python `requests`包。它可以让我们轻松执行GET、POST以及其他HTTP请求。想要安装该包，可以使用`pip install treq`，不过它已经在我们的开发机中预先安装好了。

我们更倾向于选择`treq`而不是Scrapy的`Request/crawler.engine.download()`的原因是，虽然它们都很简单，但是在性能上`treq`更有优势，我们将会在第10章中看到更详细的介绍。

9.1.2 用于写入Elasticsearch的管道

首先，我们要编写一个将Item 存储到**ES** （**Elasticsearch** ）服务器的爬虫。你可能会觉得从ES开始，甚至先于MySQL，作为持久化机制进行讲解有些不太寻常，不过其实它是我们可以做的最简单的事情。ES可以是无模式的，也就是说无需任何配置就能够使用它。对于我们这个（非常简单的）用例来说，treq 也已经足够使用。如果想要使用更高级的ES功能，则需要考虑使用txes2 或其他Python/Twisted ES包。

在我们的开发机中，已经包含正在运行的ES服务器了。下面登录到开发机中，验证其是否正在正常运行。

```
$ curl http://es:9200

{

  "name" : "Living Brain",

  "cluster_name" : "elasticsearch",

  "version" : { ... },
```

```
  "tagline" : "You Know, for Search"

}
```

在宿主机浏览器中，访问`http://localhost:9200`，也可以看到同样的结果。当访问`http://localhost:9200/properties/property/_search`时，可以看到返回的响应表示ES进行了全局性的尝试，但是没有找到任何与房产信息相关的索引。恭喜你，刚刚已经使用了ES的REST API。



在本章，我们将在properties集合中插入房产信息。你可能需要重置properties集合，此时可以使用`curl`执行DELETE请求：

```
$ curl -XDELETE http://es:9200/properties
```



本章中管道实现的完整代码包含很多额外的细节，如更多的错误处理等，不过我将通过凸显关键点的方式，保持这里的代码简洁。



本章在ch09 目录当中，其中本示例的代码为ch09/properties/properties/pipelines/es.py 。

从本质上说，爬虫代码只包含如下4行。

```
@defer.inlineCallbacks
def process_item(self, item, spider):
    data = json.dumps(dict(item), ensure_ascii=False).encode("utf-
8")
    yield treq.post(self.es_url, data)
```

其中，前两行用于定义标准的process_item() 方法，可以在其中yield 延迟操作（参考第8章）。

第 3 行用于准备要插入的data 。首先，我们将Item 转化为字典。然后使用json.dumps() 将其编码为JSON格式。ensure_ascii=False 的目的是通过不转义非ASCII字符，使得输出更加紧凑。然后，将这些JSON字符串编码为UTF-8，即JSON标准中的默认编码。

最后一行使用treq 的post() 方法执行POST请求，将文档插入到ElasticSearch中。es_url 存储在settings.py 文件当中（ES_PIPELINE_URL 设置），如http://es:9200/properties/property ，可以提供一些基本信息，如ES服务器的IP和端口（es:9200 ）、集合名称（properties ）以及想要写入的对象类型（property ）。

要想启用该管道，需要将其添加到settings.py 文件的ITEM_PIPELINES 设置当中，并且使用ES_PIPELINE_URL 设置进行初始化。

```
ITEM_PIPELINES = {
    'properties.pipelines.tidyup.TidyUp': 100,
    'properties.pipelines.es.EsWriter': 800,
}
ES_PIPELINE_URL = 'http://es:9200/properties/property'
```

完成上述工作后，我们可以进入到适当的目录当中。

```
$ pwd
```

```
/root/book/ch09/properties

$ ls

properties scrapy.cfg
```

然后，开始运行爬虫。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90

...

INFO: Enabled item pipelines: EsWriter...
```

```
INFO: Closing spider (closespider_itemcount)...

   'item_scraped_count': 106,
```

如果现在再次访问http://localhost:9200/properties/property/_ search，可以在响应的hits/total 字段中看到已经插入的条目数量，以及前10条结果。我们还可以通过添加?size=100 参数取得更多结果。在搜索URL中添加q= 参数时，可以在全部或特定字段中搜索指定关键词。最相关的结果将会出现在最前面。例如，http://localhost: 9200/properties/property/_search?q=title: london，将会返回标题中包含"London"的房产信息。对于更加复杂的查询，可以查阅 ES 的官方文档，网址为：https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html 。

ES不需要配置的原因是它可以根据我们提供的第一个属性自动检测模式（字段类型）。通过访问http://localhost:9200/properties/，可以看到其自动检测的映射关系。

让我们快速查看一下性能，使用上一章结尾处给出的方式重新运行`scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000`。平均延时从0.78秒增长到0.81秒，这是因为管道的平均时间从0.12秒增长到了0.15秒。吞吐量仍然保持在每秒大约25个Item。



使用管道将Item插入到数据库当中是不是一个好主意呢？答案是否定的。通常情况下，数据库提供的批量插入条目的方式可以有几个数量级的效率提升，因此我们应当使用这种方式。也就是说，应当将Item打包批量插入，或在爬虫结束时以后置处理的步骤执行插入。我们将在最后一章中看到这些方法。不过，许多人仍然使用Item管道插入数据库，此时使用Twisted API而不是通用/阻塞的方法实现该方案才是正确的方式。

9.1.3　使用Google Geocoding API实现地理编码的管道

每个房产信息都有地区名称，因此我们想对其进行地理编码，也就是说找到它们对应的坐标（经度、纬度）。我们可以使用这些坐标将房产信息放到地图上，或是根据它们到某个位置的距离对搜索结果进行排序。开发这种功能需要复杂的数据库、文本匹配以及空间计算。而使用Google的Geocoding API，可以避免上面提到的几个问题。可以通过浏览器或`curl`打开下述URL以获取数据。

```
$ curl "https://maps.googleapis.com/maps/api/geocode/json?sensor=false&ad
```

```
dress=london"

{

   "results" : [

         ...

         "formatted_address" : "London, UK",

         "geometry" : {

            ...

            "location" : {
```

```
            "lat" : 51.5073509,
            "lng" : -0.1277583
         },
         "location_type" : "APPROXIMATE",
         ...
   ],
   "status" : "OK"
}
```

我们可以看到一个JSON对象，当搜索"location"时，可以很快发现Google提供的是伦敦中心坐标。如果继续搜索，会发现同一文档中还包含其他位置。其中，第一个坐标位置是最相关的。因此，如果存在`results[0].geometry.location`的话，它就是我们所需要的信息。

Google的Geocoding API可以使用之前用过的技术（`treq`）进行访问。只需几行，就可以找出一个地址的坐标位置（查看`pipeline`目录的`geo.py`文件），其代码如下。

```
@defer.inlineCallbacks
def geocode(self, address):
   endpoint = 'http://web:9312/maps/api/geocode/json'

   parms = [('address', address), ('sensor', 'false')]
   response = yield treq.get(endpoint, params=parms)
   content = yield response.json()

   geo = content['results'][0]["geometry"]["location"]
   defer.returnValue({"lat": geo["lat"], "lon": geo["lng"]})
```

该函数使用了一个和前面用过的URL相似的URL，不过在这里将其指向到一个假的实现，以使其执行速度更快，侵入性更小，可离线使用并且更加可预测。可以使用`endpoint =`

'https://maps.googleapis.com/maps/api/geocode/json' 来访问Google的服务器，不过需要记住的是Google对请求有严格的限制。address 和sensor 的值都通过treq 的get() 方法的params 参数进行了自动URL编码。treq.get() 方法返回了一个延迟操作，我们对其执行yield 操作，以便在响应可用时恢复它。对response.json() 的第二个yield 操作，用于等待响应体加载完成并解析为Python对象。此时，我们可以得到第一个结果的位置信息，将其格式化为字典后，使用defer.returnValue() 返回，该方法是从使用inlineCallbacks 的方法返回值的最适当的方式。如果任何地方存在问题，该方法会抛出异常，并通过Scrapy报告给我们。

通过使用geocode() ，process_item() 可以变为一行代码，如下所示。

```
item["location"] = yield self.geocode(item["address"][0])
```

我们可以在ITEM_PIPELINES 设置中添加并启用该管道，其优先级数值应当小于ES的优先级数值，以便ES获取坐标位置的值。

```
ITEM_PIPELINES = {
    ...
    'properties.pipelines.geo.GeoPipeline': 400,
```

我们启用调试数据，运行一个快速的爬虫。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=90 -L DEBUG

...

{'address': [u'Greenwich, London'],

...

 'image_urls': [u'http://web:9312/images/i06.jpg'],

 'location': {'lat': 51.482577, 'lon': -0.007659},

 'price': [1030.0],

...
```

现在，可以看到Item中包含了`location` 字段。太好了！不过当使用真实的Google API的URL临时运行它时，很快就会得到类似下面的异常。

```
File "pipelines/geo.py" in geocode (content['status'], address))
Exception: Unexpected status="OVER_QUERY_LIMIT" for
address="*London"
```

这是我们在完整代码中放入的一个检查，用于确保Geocoding API的响应中`status` 字段的值是`OK` 。如果该值非真，则说明我们得到的返回数据不是期望的格式，无法被安全使用。在本例中，我们得到了`OVER_QUERY_LIMIT` 状态，可以清楚地说明在什么地方发生了错误。这可能是我们在许多案例中都会面临的一个重要问题。由于Scrapy的引擎具备较高的性能，缓存和资源请求的限流成为了必须考虑的问题。

可以访问Geocoder API的文档来了解其限制："免费用户API：每24小时允许2500个请求，每秒允许5个请求"。即使使用了Google Geocoding API的付费版本，仍然会有每秒10个请求的限流，这就意味着该讨论仍然是有意义的。



下面的实现看起来可能会比较复杂，但是它们必须在上下文中进行判断。而在典型的多线程环境中创建此类组件需要线程池和同步，这样就会产生更加复杂的代码。

下面是使用Twisted技术实现的一个简单而又足够好用的限流引擎。

```
class Throttler(object):
    def __init__(self, rate):
        self.queue = []
        self.looping_call = task.LoopingCall(self._allow_one)
        self.looping_call.start(1. / float(rate))

    def stop(self):
        self.looping_call.stop()

    def throttle(self):
        d = defer.Deferred()
        self.queue.append(d)
        return d

    def _allow_one(self):
        if self.queue:
            self.queue.pop(0).callback(None)
```

该代码中，延迟操作排队进入列表中，每次调用`_allow_one()`时依次触发它们；`_allow_one()`检查队列是否为空，如果不是，则调用最旧的延迟操作的`callback()`（先入先出，FIFO）。我们使用Twisted的`task.LoopingCall()` API周期性调用`_allow_one()`。使

用Throttler 非常简单。我们可以在管道的__init__ 中对其进行初始化，并在爬虫结束时对其进行清理。

```
class GeoPipeline(object):
    def __init__(self, stats):
        self.throttler = Throttler(5) # 5 Requests per second

    def close_spider(self, spider):
        self.throttler.stop()
```

在使用想要限流的资源之前（在本例中为在process_item() 中调用geocode() ），需要对限流器的throttle() 方法执行yield 操作。

```
yield self.throttler.throttle()
item["location"] = yield self.geocode(item["address"][0])
```

在第一个yield 时，代码将会暂停，等待足够的时间过去之后再恢复。比如，某个时刻共有11个延迟操作在队列中，我们的速率限制是每秒5个请求，我们的代码将会在队列清空时恢复，大约为11/5=2.2秒。

使用Throttler 后，我们不再会发生错误，但是爬虫速度会变得非常慢。通过观察发现，示例的房产信息中只有有限的几个不同位置。这是使用缓存的一个非常好的机会。我们可以使用一个简单的Python字典来实现缓存，不过这种情况下将会产生竞态条件，导致不正确的API调用。下面是一个没有该问题的缓存，此外还演示了一些Python和Twisted的有趣特性。

```
class DeferredCache(object):
    def __init__(self, key_not_found_callback):
        self.records = {}
        self.deferreds_waiting = {}
        self.key_not_found_callback = key_not_found_callback

    @defer.inlineCallbacks
    def find(self, key):
        rv = defer.Deferred()

        if key in self.deferreds_waiting:
            self.deferreds_waiting[key].append(rv)
        else:
            self.deferreds_waiting[key] = [rv]

            if not key in self.records:
                try:
                    value = yield self.key_not_found_callback(key)
                    self.records[key] = lambda d: d.callback(value)
                except Exception as e:
                    self.records[key] = lambda d: d.errback(e)

            action = self.records[key]
            for d in self.deferreds_waiting.pop(key):
                reactor.callFromThread(action, d)

        value = yield rv
        defer.returnValue(value)
```

该缓存看起来和人们通常期望的有些不同。它包含两个组成部分。

- self.deferreds_waiting：这是一个延迟操作的队列，等待指定键的值。
- self.records：这是已经出现的键-操作对的字典。

如果查看find()实现的中间部分，就会发现如果没有在self.records中找到一个键，则会调用一个预定义的callback函

数，取得缺失值（yield self.key_not_found_callback(key) ）。该回调函数可能会抛出一个异常。我们要如何在Python中以紧凑的方式存储这些值或异常呢？由于Python是一种函数式语言，我们可以根据是否出现异常，在self.records 中存储调用延迟操作的callback 或errback 的小函数（lambda ）。在定义时，该值或异常被附加到lambda 函数中。函数中对变量的依赖被称为闭包，这是大多数函数式编程语言最显著和强大的特性之一。



缓存异常有些不太常见，不过这意味着如果在第一次查找某个键时，key_not_found_callback(key) 抛出了异常，那么接下来对相同键再次查询时仍然会抛出同样的异常，不需要再执行额外的调用。

find() 实现的剩余部分提供了避免竞态条件的机制。如果要查询的键已经在进程当中，将会在self.deferreds_waiting 字典中有记录。在这种情况下，我们不再额外调用key_not_found_callback() ，只是添加到延迟操作列表中，等待该键。当key_not_found_callback() 返回，并且该键的值变为可用时，触发每个等待该键的延迟操作。我们可以直接执行action(d) ，而不是使用reactor.callFromThread() ，不过这样就必须处理所有抛出的异常，并且会创建一个不必要的长延迟链。

使用缓存非常简单。只需在__init__() 中对其初始化，并在执行API 调用时设置回调函数即可。在process_item() 中，按照如下代

码使用缓存。

```
def __init__(self, stats):
    self.cache = DeferredCache(self.cache_key_not_found_callback)

@defer.inlineCallbacks
def cache_key_not_found_callback(self, address):
    yield self.throttler.enqueue()
    value = yield self.geocode(address)
    defer.returnValue(value)

@defer.inlineCallbacks
def process_item(self, item, spider):
    item["location"] = yield self.cache.find(item["address"][0])
    defer.returnValue(item)
```

本例的完整代码包含了更多的错误处理代码，能够对限流导致的错误重试调用（一个简单的`while`循环），并且还包含了更新爬虫状态的代码。



本例的完整代码文件地址为：`ch09/properties/properties/pipelines/geo2.py`。

要想启用该管道，需要禁用（注释掉）之前的实现，并且在`settings.py`文件的`ITEM_PIPELINES`中添加如下代码。

```
ITEM_PIPELINES = {
    'properties.pipelines.tidyup.TidyUp': 100,
    'properties.pipelines.es.EsWriter': 800,
```

```
    # DISABLE 'properties.pipelines.geo.GeoPipeline': 400,
    'properties.pipelines.geo2.GeoPipeline': 400,
}
```

然后，可以按照如下代码运行该爬虫。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000

...

Scraped... 15.8 items/s, avg latency: 1.74 s and avg time in pipelines:

0.94 s

Scraped... 32.2 items/s, avg latency: 1.76 s and avg time in pipelines:

0.97 s
```

```
Scraped... 25.6 items/s, avg latency: 0.76 s and avg time in pipelines:
0.14 s
...
: Dumping Scrapy stats:...
   'geo_pipeline/misses': 35,
   'item_scraped_count': 1019,
```

可以看到，爬取延时最初由于填充缓存的原因非常高，但是很快就回到了之前的值。统计显示总共有35次未命中，这正是我们所用的示例

数据集内不同位置的数量。显然，在本例中总共有1019 - 35 = 984次命中缓存。如果使用真实的Google API，并将每秒对API的请求数量稍微增加，比如通过将`Throttler(5)`改为`Throttler(10)`，把每秒请求数从5增加到10，就会在`geo_pipeline/retries`统计中得到重试的记录。如果发生任何错误，比如使用API无法找到一个位置，将会抛出异常，并且会在`geo_pipeline/errors`统计中被捕获到。如果某个位置的坐标已经被设置（后面的小节中看到），则会在`geo_pipeline/already_set`统计中显示。最后，当访问`http://localhost:9200/ properties/ property/_search`，查看房产信息的ES时，可以看到包含坐标位置值的条目，比如`{..."location": {"lat": 51.5269736, "lon": -0.0667204}...}`，这和我们所期望的一样（在运行之前清理集合，确保看到的不是旧值）。

9.1.4 在Elasticsearch中启用地理编码索引

既然已经拥有了坐标位置，现在就可以做一些事情了，比如根据距离对结果进行排序。下面是一个HTTP POST请求（使用`curl`执行），返回标题中包含"Angel"的房产信息，并按照它们与点`{51.54, -0.19}`的距离进行排序。

```
$ curl http://es:9200/properties/property/_search -d '{

   "query" : {"term" : { "title" : "angel" } },
```

```
    "sort": [{"_geo_distance": {

        "location":     {"lat": 51.54, "lon": -0.19},

        "order":        "asc",

        "unit":         "km",

        "distance_type": "plane"

}}]}'
```

唯一的问题是当尝试运行它时，会发现运行失败，并得到了一个错

误信息："failed to find mapper for [location] for geo distance based sort"。这说明位置字段并不是执行空间操作的适当格式。要想设置为合适的类型，则需要手动重写其默认类型。首先，将其自动检测的映射关系保存到文件中。

```
$ curl 'http://es:9200/properties/_mapping/property' > property.txt
```

然后编辑property.txt 的如下代码。

```
"location":{"properties":{"lat":{"type":"double"},"lon":{"type":"d
ouble"}}}
```

将该行的代码修改为如下代码。

```
"location": {"type": "geo_point"}
```

另外，我们还删除了文件尾部的{"properties":{"mappings": and two }}。对该文件的修改到此为止。现在可以按如下代码删除旧

类型，使用指定的模式创建新类型。

```
$ curl -XDELETE 'http://es:9200/properties'

$ curl -XPUT 'http://es:9200/properties'

$ curl -XPUT 'http://es:9200/properties/_mapping/property' --data

@property.txt
```

现在可以再次运行该爬虫，并且可以重新运行本节前面的`curl`命令，此时将会得到按照距离排序的结果。我们的搜索返回了房产信息的JSON，额外包含了一个`sort`字段，该字段的值是到搜索点的距离，单位为千米。

9.2　与标准Python客户端建立数据库接口

有很多重要的数据库遵从Python数据库API规范2.0版本，包括MySQL、PostgreSQL、Oracle、Microsoft SQL Server和SQLite。它们的驱动一般都比较复杂且久经考验，如果为Twisted重新实现的话则是巨大的浪费。人们可以在Twisted应用中使用这些数据库客户端，比如在Scrapy使用`twisted.enterprise.adbapi`库。我们将使用MySQL作为示例演示其使用，不过对于任何其他兼容的数据库来说，也可以应用相同的原则。

9.2.1　用于写入MySQL的管道

MySQL是一个非常强大且流行的数据库。我们将编写一个管道，将item写入到其中。我们已经在虚拟环境中运行了一个MySQL实例。现在只需使用MySQL命令行工具执行一些基本管理即可，同样该工具也已经在开发机中预安装好了，下面执行如下操作打开MySQL控制台。

```
$ mysql -h mysql -uroot -ppass
```

这将会得到MySQL的提示符，即`mysq>`，现在可以创建一个简单的数据库表，其中包含一些字段，如下所示。

```
mysql> create database properties;
```

```
mysql> use properties

mysql> CREATE TABLE properties (

  url varchar(100) NOT NULL,

  title varchar(30),

  price DOUBLE,

  description varchar(30),

  PRIMARY KEY (url)

);
```

```
mysql> SELECT * FROM properties LIMIT 10;

Empty set (0.00 sec)
```

非常好，现在拥有了一个MySQL数据库，以及一张名为properties 的表，其中包含了一些字段，此时可以准备创建管道了。请保持MySQL的控制台为开启状态，因为之后还会回来检查是否正确插入了值。如果想退出控制台，只需要输入exit 即可。



在本节，我们将会向MySQL数据库中插入房产信息。如果你想擦除它们，可以使用如下命令：

```
mysql> DELETE FROM properties;
```

我们将使用Python的MySQL客户端。我们还将安装一个名为`dj-database-url`的小工具模块，帮助我们解析连接的URL（仅用于为我们在IP、端口、密码等不同设置中切换节省时间）。可以使用`pip install dj-database-url MySQL-python`安装这两个库，不过我们已经在开发环境中安装好它们了。我们的MySQL管道非常简单，如下所示。

```
from twisted.enterprise import adbapi
...
class MysqlWriter(object):
    ...
    def __init__(self, mysql_url):
        conn_kwargs = MysqlWriter.parse_mysql_url(mysql_url)
        self.dbpool = adbapi.ConnectionPool('MySQLdb',
                                            charset='utf8',
                                            use_unicode=True,
                                            connect_timeout=5,
                                            **conn_kwargs)

    def close_spider(self, spider):
        self.dbpool.close()

    @defer.inlineCallbacks
    def process_item(self, item, spider):
        try:
```

```
            yield self.dbpool.runInteraction(self.do_replace, item)
        except:
            print traceback.format_exc()

        defer.returnValue(item)

    @staticmethod
    def do_replace(tx, item):
        sql = """REPLACE INTO properties (url, title, price,
        description) VALUES (%s,%s,%s,%s)"""

        args = (
            item["url"][0][:100],
            item["title"][0][:30],
            item["price"][0],
            item["description"][0].replace("\r\n", " ")[:30]
        )

        tx.execute(sql, args)
```



本示例的完整代码地址为ch09/properties/properties/pipeline/mysql.py。

本质上，大部分代码仍然是模板化的爬虫代码。我们省略的代码用于将MYSQL_PIPELINE_URL 设置中包含的mysql://user:pass@ip/database 格式的URL解析为独立参数。在爬虫的__init__() 中，我们将这些参数传给adbapi.ConnectionPool()，使用adbapi 的基础功能初始化MySQL连接池。第一个参数是想要导入的模块名称。在该MySQL示例中，

为MySQLdb 。我们还为MySQL客户端设置了一些额外的参数，用于处理Unicode和超时。所有这些参数会在每次adbapi 需要打开新连接时，前往底层的MySQLdb.connect() 函数。当爬虫关闭时，我们为该连接池调用close() 方法。

我们的process_item() 方法实际上包装了dbpool.runInteraction() 。该方法将稍后调用的回调方法放入队列，当来自连接池的某个连接的Transaction 对象变为可用时，调用该回调方法。Transaction 对象的API与DB-API游标相似。在本例中，回调方法为do_replace() ，该方法在后面几行进行了定义。@staticmethod 意味着该方法指向的是类，而不是具体的类实例，因此，可以省略平时使用的self 参数。当不使用任何成员时，将方法静态化是个好习惯，不过即使忘记这么做，也没有问题。该方法准备了一个SQL字符串和几个参数，调用Transaction 的execute() 方法执行插入。我们的SQL语句使用了REPLACE INTO 来替换已经存在的条目，而不是更常见的INSERT INTO ，原因是如果条目已经存在，可以使用相同的主键。在本例中这种方式非常便捷。如果想使用SQL返回数据，如SELECT 语句，可以使用dbpool.runQuery() 。如果想要修改默认游标，可以通过设置adbapi.ConnectionPool() 的cursorclass 参数来实现，比如设置cursorclass=MySQLdb.cursors.DictCursor ，可以让数据获取更加便捷。

要想使用该管道，需要在settings.py 文件的ITEM_PIPELINES 字典中添加它，另外还需要设置MYSQL_PIPELINE_URL 属性。

```
ITEM_PIPELINES = { ...
```

```
    'properties.pipelines.mysql.MysqlWriter': 700,
...
MYSQL_PIPELINE_URL = 'mysql://root:pass@mysql/properties'
```

执行如下命令。

```
scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000
```

该命令运行后，可以回到MySQL提示符下，按如下方式查看数据库中的记录。

```
mysql> SELECT COUNT(*) FROM properties;

+----------+

| 1006 |
```

```
+----------+

mysql> SELECT * FROM properties LIMIT 4;

+----------------+----------------------+-------+----------+

| url            | title                | price | description

+----------------+----------------------+-------+----------+

| http://...0.html | Set Unique Family Well   | 334.39 | website c

| http://...1.html | Belsize Marylebone Shopp | 388.03 | features

| http://...2.html | Bathroom Fully Jubilee S | 365.85 | vibrant own
```

```
| http://...3.html | Residential Brentford Ot | 238.71 | go court

+------------------+------------------------+--------+-----------+

4 rows in set (0.00 sec)
```

延时和吞吐量等性能和之前保持相同，相当不错。

9.3　使用Twisted专用客户端建立服务接口

到目前为止，我们看到了如何通过`treq` 使用类REST API。Scrapy还可以和许多其他使用Twisted专用客户端的服务建立接口。比如，我们想要与MongoDB建立接口，当搜索"MongoDB Python"时，将会得到`PyMongo`， 该库是阻塞/同步的，不能和Twisted一起使用，除非使用后续小节中的方法，在管道中描述线程，处理阻塞操作。如果搜索"MongoDB Twisted Python"，将会得到`txmongo`， 该库可以在Twisted

和Scrapy中完美运行。通常情况下，Twisted客户端背后的社区都很小，但相比自行编写客户端，这仍然是一个更好的选择。我们将使用一个类似的Twisted专用客户端作为接口，处理Redis键值对存储。

9.3.1　用于读写Redis的管道

Google Geocoding API是按照IP进行限制的。我们可以利用多个IP（例如使用多台服务器）进行缓解，此时需要避免重复请求其他机器上已经完成地理编码的地址。这种情况也适用于之前运行中曾见到过的地址。我们不想浪费宝贵的限额。



请与API供应商沟通，确保在他们的策略下这种做法是可行的。比如，你可能必须每隔几分钟/小时就要丢弃掉缓存记录，或者根本不允许缓存。

我们可以使用Redis的键值对缓存，从本质上说，它是一个分布式的字典。我们已经在vagrant环境中运行了一个Redis实例，可以使用`redis-cli`命令，从开发机连接它并执行基本操作。

```
$ redis-cli -h redis

redis:6379> info keyspace
```

```
# Keyspace

redis:6379> set key value

OK

redis:6379> info keyspace

# Keyspace

db0:keys=1,expires=0,avg_ttl=0

redis:6379> FLUSHALL

OK
```

```
redis:6379> info keyspace

# Keyspace

redis:6379> exit
```

通过Google搜索"Redis Twisted"，我们找到了txredisapi 库。其本质区别是它不再是同步Python库的包装，而是适用于Twisted的库，它使用reactor.connectTCP() 连接Redis、实现Twisted协议等。使用该库的方式与其他库类似，不过在Twisted应用中使用它时，其效率肯定会更高一些。我们在安装它时可以再附带一个工具库——dj_redis_url ，该工具库用于解析Redis配置URL，我们可以使用pip 进行安装（sudo pip install txredisapi dj_redis_url ），和往常一样，在我们的开发机中也已经预先安装好了这些库。

可以按如下代码初始化RedisCache 。

```
from txredisapi import lazyConnectionPool
```

```
class RedisCache(object):
...
    def __init__(self, crawler, redis_url, redis_nm):
        self.redis_url = redis_url
        self.redis_nm = redis_nm

        args = RedisCache.parse_redis_url(redis_url)
        self.connection = lazyConnectionPool(connectTimeout=5,
                                             replyTimeout=5,
                                             **args)
        crawler.signals.connect(
                self.item_scraped,signal=signals.item_scraped)
```

该管道非常简单。为了连接Redis服务器，我们需要主机地址、端口等参数，由于这些参数是以URL格式存储的，因此需要使用parse_redis_url()方法解析该格式（为简洁起见已经省略）。为键设置前缀作为命名空间的行为非常常见，在本例中，我们将其存储在redis_nm中。然后，使用txredisapi的lazyConnectionPool()，打开到服务器的连接。

最后一行使用了一个很有意思的函数。我们的目的是将地理编码管道与该管道包装起来。如果在Redis中没有某个值，我们将不会设置该值，我们的地理编码管道将像之前那样使用API对地址进行地理编码。在该操作完成之后，需要有一种方式在Redis中缓存这些键值对，在这里是通过连接到signals.item_scraped信号的方式实现的。我们定义的回调（item_scraped()方法，将很快看到）在非常靠后的位置被调用，此时坐标位置将会被设置。



本示例的完整代码位于ch09/properties/properties/pipelines/redis.py。

我们通过查找和记录每个Item 的地址和位置，保持了缓存的简单性。这对Redis来说是很有意义的，因为它经常运行在同一个服务器当中，这使得它运行速度非常快。如果不是这种情况，那么可能需要添加一个基于字典的缓存，与我们在地理编码管道中的实现类似。下面是处理传入的Item的方法。

```
@defer.inlineCallbacks
def process_item(self, item, spider):
    address = item["address"][0]
    key = self.redis_nm + ":" + address
    value = yield self.connection.get(key)
    if value:
        item["location"] = json.loads(value)
    defer.returnValue(item)
```

和大家的期望相同。我们得到了地址，为其添加前缀，然后使用txredisapi connection 的get() 方法在Redis中查询。我们在Redis中存储的值是JSON编码的对象。如果值已经设定，则使用JSON对其进行解码，并将其设为坐标位置。

当一个Item 到达所有管道的结尾时，我们重新捕获它，确保存储到Redis的位置值当中。下面是实现代码。

```
from txredisapi import ConnectionError

def item_scraped(self, item, spider):
    try:
        location = item["location"]
        value = json.dumps(location, ensure_ascii=False)
    except KeyError:
        return

    address = item["address"][0]
    key = self.redis_nm + ":" + address
    quiet = lambda failure: failure.trap(ConnectionError)
    return self.connection.set(key, value).addErrback(quiet)
```

这里同样没有什么惊喜。如果我们找到一个位置，就可以得到地址，为其添加前缀，并使用它们作为键值对，用于txredisapi 连接的set() 方法。你会发现该函数没有使用@defer.inlineCallbacks，这是因为在处理signals.item_scraped 时并不支持该装饰器。这就意味着无法再对connection.set() 使用非常便捷的yield 操作，不过我们可以做的工作是返回延迟操作，Scrapy可以用它串联任何未来的信号进行监听。无论何种情况，如果到Redis的连接无法执行connection.set()，就会抛出一个异常。可以通过添加自定义错误处理到connection.set() 返回的延迟操作中，静默忽略该异常。在该错误处理中，我们将失败作为参数传递，并告知它们对任何ConnectionError 执行trap() 操作。这是Twisted的延迟操作API的一个非常好用的功能。通过在预期的异常中使用trap()，我们能够以紧凑的方式静默忽略它们。

为了启用该管道，我们所需做的就是将其添加到ITEM_PIPELINES

设置中，并在settings.py 文件中提供一个REDIS_PIPELINE_URL 。为该管道设置一个比地理编码管道更小的优先级值非常重要，否则其运行就会太迟，无法起到作用。

```
ITEM_PIPELINES = { ...
    'properties.pipelines.redis.RedisCache': 300,
    'properties.pipelines.geo.GeoPipeline': 400,
...
REDIS_PIPELINE_URL = 'redis://redis:6379'
```

我们可以像平时那样运行该爬虫。第一次运行将会和之前类似，不过接下来的每次运行都会像下面这样。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=100

...

INFO: Enabled item pipelines: TidyUp, RedisCache, GeoPipeline,

MysqlWriter, EsWriter
```

```
...
Scraped... 0.0 items/s, avg latency: 0.00 s, time in pipelines: 0.00 s
Scraped... 21.2 items/s, avg latency: 0.78 s, time in pipelines: 0.15 s
Scraped... 24.2 items/s, avg latency: 0.82 s, time in pipelines: 0.16 s
...
INFO: Dumping Scrapy stats: {...
   'geo_pipeline/already_set': 106,
   'item_scraped_count': 106,
```

可以看到GeoPipeline和RedisCache都已经启用，并且RedisCache会首先进行。另外，还可以注意到geo_pipeline/already_set统计值是106。这些是GeoPipeline从Redis缓存中找到的预先填充好的item，并且它们都不需要请求Google API调用。如果Redis缓存为空，你会看到一些键依然会使用Google API进行处理。在性能方面，我们注意到GeoPipeline引发的初始行为现在没有了。实际上，由于目前使用了缓存，因此绕过了每秒5个请求的API限制。当使用Redis时，还应当考虑使用过期键，使系统可以周期性地刷新缓存数据。

9.4　为CPU密集型、阻塞或遗留功能建立接口

本章最后一节讨论的是访问大多数非Twisted的工作。尽管有高效的异步代码所带来的巨大收益，但为Twisted和Scrapy重写每个库，既不现实也不可行。使用Twisted的线程池和reactor.spawnProcess()方法，我们可以使用任何Python库甚至其他语言编写的二进制包。

9.4.1　处理CPU密集型或阻塞操作的管道

第8章讲到，reactor对于简短、非阻塞的任务非常理想。如果必须要执行一些更复杂或是涉及阻塞的事情，该怎么做呢？Twisted提供了线程池，可以使用reactor.callInThread() API调用，在一些线程中

执行慢操作，而不是在主线程中执行（Twisted的reactor）。这就意味着reactor会持续运行其处理过程，并在计算发生时响应事件。请注意，在线程池中的处理不是线程安全的。这就是说当你使用全局状态时，又会出现多线程编程中所有的传统同步问题。让我们从该管道的一个简单版本起步，逐渐编写出完整的代码。

```
class UsingBlocking(object):
    @defer.inlineCallbacks
    def process_item(self, item, spider):
        price = item["price"][0]

        out = defer.Deferred()
        reactor.callInThread(self._do_calculation, price, out)
        item["price"][0] = yield out

        defer.returnValue(item)

    def _do_calculation(self, price, out):
        new_price = price + 1
        time.sleep(0.10)
        reactor.callFromThread(out.callback, new_price)
```

在前面的管道中，我们看到了实际运行的基本原语。对于每个`Item`，我们抽取其价格，并希望使用`_do_calculation()`方法处理它。该方法使用了一个阻塞操作`time.sleep()`。我们将使用`reactor.callInThread()`调用把它放到另一个线程中运行。其中，被调用的函数以及传给该函数的任意数量的参数将会作为参数。显然，我们不只传递了`price`，还创建并传递了一个名为`out`的延迟操作。当`_do_ calculation()`完成计算时，我们将使用out回调返回值。在下一步中，我们对这个延迟操作执行了yield处理，并为价格设置

了新值，最后返回Item 。

在_do_ calculation() 中，注意到有一个简单的计算——价格自增1，然后是100毫秒的睡眠。这是非常多的时间，如果在reactor线程中调用，它将使我们每秒处理的页数无法超过10页。通过使其在其他线程中运行，就不再有这个问题了。任务将会在线程池中排队，等待出现可用的线程，一旦进入线程执行，该线程就将睡眠100毫秒。最后一步是触发out 回调。正常情况下，可以使用out.callback(new_price) ，不过由于现在处于另一个线程中，这种方法不再安全。如果这样做，会导致延迟操作的代码和Scrapy的功能会从另一个线程调用，迟早会出现错误的数据。替代方案是使用reactor.callFromThread() ，同样，也是将函数作为参数，并将任意数量的额外参数传到函数中。该函数将会排队，由reactor线程调用；而另一方面，会解除process_item() 对象yield 操作的阻塞，为该Item 恢复Scrapy操作。

如果有全局状态（比如计数器、移动平均值等）的话，那么在_do_calculation() 中使用它们会发生什么呢？例如，我们添加两个变量——beta 和delta ，如下所示。

```
class UsingBlocking(object):
    def __init__(self):
        self.beta, self.delta = 0, 0
    ...
    def _do_calculation(self, price, out):
        self.beta += 1
        time.sleep(0.001)
        self.delta += 1
        new_price = price + self.beta - self.delta + 1
        assert abs(new_price-price-1) < 0.01

        time.sleep(0.10)...
```

上面的代码存在问题，我们会得到断言错误。这是因为如果一个线程在`self.beta`和`self.delta`之间切换，而另一个线程使用这些`beta/delta`的值恢复计算价格，那么会发现它们处于不一致的状态（`beta`比`delta`大），因此，会计算出错误的结果。短暂的睡眠使该问题更容易产生，不过即便没有它，竞态条件也将很快出现。为了避免此类问题发生，必须使用锁，比如使用Python的`threading.RLock()`递归锁。当使用锁时，我们可以确信不会存在两个线程同时执行其保护的临界区的情况。

```
class UsingBlocking(object):
    def __init__(self):
        ...
        self.lock = threading.RLock()
    ...
    def _do_calculation(self, price, out):
        with self.lock:
            self.beta += 1
            ...
            new_price = price + self.beta - self.delta + 1

        assert abs(new_price-price-1) < 0.01 ...
```

前面的代码现在是正确的。请记住我们并不需要保护整段代码，只需覆盖全局状态的使用就够了。



本示例的完整代码位于ch09/properties/p``roperties/pipelines/computation.py 文件中。

要想使用该管道，只需在settings.py 文件中将其添加到ITEM_PIPELINES 设置即可，如下所示。

```
ITEM_PIPELINES = { ...
    'properties.pipelines.computation.UsingBlocking': 500,
```

可以按照平时那样运行该爬虫。按照预期，管道延时显著增长了100毫秒，不过我们惊喜地发现吞吐量几乎保持不变，即每秒25个item左右。

9.4.2　使用二进制或脚本的管道

对于一个遗留功能来说，最不可知的接口就是独立的可执行程序或脚本。它可能需要几秒钟时间启动（比如从数据库中加载数据），不过在这之后，它可能会在一小段延时内处理许多值。即使对于这种情况，Twisted仍然能够覆盖。我们可以使用reactor.spawnProcess() API以及相关的protocol.ProcessProtocol 运行任何类型的可执行程序。来看一个例子，该示例的脚本如下所示。

```
#!/bin/bash

trap "" SIGINT

sleep 3

while read line

do

    # 4 per second

    sleep 0.25

    awk "BEGIN {print 1.20 * $line}"
```

```
done
```

这是一个简单的bash脚本。当它启动后，会禁用*Ctrl* + *C* 。这是为了解决*Ctrl* + *C* 派生到子进程后过早终止，导致Scrapy自身无法停止，无限等待子进程返回结果的系统特性。禁用*Ctrl* + *C* 后，脚本将会睡眠3秒钟，以模拟启动时间。然后脚本会从输入中读取行，等待250毫秒，再返回结果价格，该计算使用Linux的`awk` 命令将原值乘以1.2倍。该脚本的最大吞吐量是每秒4个`Item` 。可以使用一个简短的会话对其进行测试，如下所示。

```
$ properties/pipelines/legacy.sh

12 <- If you type this quickly you will wait ~3 seconds to get results

14.40
```

```
13 <- For further numbers you will notice just a slight delay



15.60
```

由于*Ctrl* + *C*被禁用，我们必须使用*Ctrl* + *D*终止会话。不错！那么，我们要如何在Scrapy中使用该脚本呢？仍然从一个简化的版本起步。

```
class CommandSlot(protocol.ProcessProtocol):
    def __init__(self, args):
        self._queue = []
        reactor.spawnProcess(self, args[0], args)

    def legacy_calculate(self, price):
        d = defer.Deferred()
        self._queue.append(d)
        self.transport.write("%f\n" % price)
        return d

    # Overriding from protocol.ProcessProtocol
    def outReceived(self, data):
        """Called when new output is received"""
        self._queue.pop(0).callback(float(data))

class Pricing(object):
    def __init__(self):
        self.slot = CommandSlot(['properties/pipelines/legacy.sh'])
```

```
    @defer.inlineCallbacks
    def process_item(self, item, spider):
        item["price"][0] = yield self.slot.legacy_calculate(item["price"][0
])
        defer.returnValue(item)
```

我们可以在这里找到名为CommandSlot 的ProcessProtocol 的定义，以及Pricing 爬虫。在__init__() 中，我们创建了新的CommandSlot，其构造方法初始化了一个空队列，并使用reactor.spawnProcess() 启动了一个新的进程。该调用将从进程中传输和接收数据的ProcessProtocol 作为第一个参数。在本例中，该值为self，因为spawnProcess() 是在protocol 类中进行调用的。第二个参数是可执行程序的名称。第三个参数args 将该二进制程序的所有命令行参数作为字符串列表保留。

在管道的process_item() 中，基本上将所有工作都委托给CommandSlot 的legacy_calculate() 方法，它将返回一个延迟操作，并执行yield 操作。legacy_calculate() 创建了一个延迟操作，使其排队，然后使用transport.write() 将价格写入到进程当中。transport 由ProcessProtocol 提供，用于让我们和进程进行通信。无论我们何时从进程中接收到数据，都会调用outReceived()。通过延迟操作排队，以及按顺序处理的shell脚本，我们可以从队列中只弹出最旧的延迟操作，使用接收到的值触发它。到此为止。我们可以通过在ITEM_PIPELINES 中添加它的方式，启动该管道，并像平时那样运行。

```
ITEM_PIPELINES = {...
    'properties.pipelines.legacy.Pricing': 600,
```

如果我们运行一次，就会发现其性能非常糟糕。如我们所料，我们的处理成为瓶颈，限制了吞吐量只能达到每秒4个Item。要想增长吞吐量，我们所能做的就是对管道进行一些修改，允许该类并行运行多个，如下所示。

```
class Pricing(object):
    def __init__(self):
        self.concurrency = 16
        args = ['properties/pipelines/legacy.sh']
        self.slots = [CommandSlot(args)
                      for i in xrange(self.concurrency)]
        self.rr = 0

    @defer.inlineCallbacks
    def process_item(self, item, spider):
        slot = self.slots[self.rr]
        self.rr = (self.rr + 1) % self.concurrency
        item["price"][0] = yield
                         slot.legacy_calculate(item["price"][0])
        defer.returnValue(item)
```

我们将其修改为启动16个实例，并以轮询的方式为每个实例发送价格。该管道现在提供了每秒16×4 = 64个item的吞吐量。我们可以通过一个快速爬取来确认，如下所示。

```
$ scrapy crawl easy -s CLOSESPIDER_ITEMCOUNT=1000
```

```
...

Scraped... 0.0 items/s, avg latency: 0.00 s and avg time in pipelines:

0.00 s

Scraped... 21.0 items/s, avg latency: 2.20 s and avg time in pipelines:

1.48 s

Scraped... 24.2 items/s, avg latency: 1.16 s and avg time in pipelines:

0.52 s
```

延时和预期一样，增长到250毫秒，不过吞吐量仍然是每秒25个item。

请注意，前面的方法中使用了`transport.write()`将shell脚本输入中的所有价格排入队列。对于你的应用而言，这种方式可能合适，也可能不合适，尤其是当它使用了更多的数据而不仅仅是几个数字时。本例完整代码会将所有值和回调排入队列，并且只有在前一次结果被接收后，才会向脚本发送新值。你会发现这种方式对你的遗留应用更加友好，不过也增添了一些复杂度。

9.5　本章小结

本章讲解了一些复杂的Scrapy管道。到目前为止，我们已经学习了Twisted编程方面所有可能需要的内容，并且知道了如何实现进程、使用Item进程管道等复杂功能。我们通过在延时和吞吐量方面添加更多管道阶段，看到了性能是如何变化的。通常情况下，延时和吞吐量被认为是成反比的，不过这是建立在常数并发的假设下的（例如线程的数例有限）。在我们的例子中，我们从N = S · T = 25 · 0.77 ≅ 19开始，在添加管道后，最终达到N = 25·3.33 ≅ 83，并且没有任何性能问题。这就是Twisted编程的力量！现在我们可以进入第10章，使Scrapy的性能更加完美。

第10章　理解Scrapy性能

通常情况下，性能很容易出现问题。对于Scrapy来说，性能就不只是容易出现问题了，而是几乎肯定会出现，因为它有很多有悖常理的行为。除非你对Scrapy内部有非常好的理解，否则你会发现，即使非常努力地优化性能，也很可能得不到收益。这是使用高性能、低延迟以及高并发环境复杂性的一部分。在优化瓶颈性能时，阿姆达尔定律仍然是正确的，不过除非你能指明真正的瓶颈所在，否则在系统其他任何部分的优化都无法增长每秒能够抓取的item数量（吞吐量）。我们可以从Goldratt博士经典的*The Goal* 一书中获得更多的感知，这本商务书籍通过优秀的隐喻对瓶颈、延迟和吞吐量的理念进行了阐释。相同的理念同样也适用于软件。本章将帮助你找出Scrapy配置中的瓶颈，以及避免出现明显的错误。

请注意本章是一个进阶章节，其中会涉及一些数学知识。计算将会比较简单，并且会附有用于展示相同概念的图表。如果你不喜欢数学，只需忽略掉公式即可，你仍然能够获得Scrapy性能如何工作的重要领悟。

10.1　Scrapy引擎——一种直观方式

并行系统看起来与管道系统很相似。在计算机科学中，我们使用队列符号来表示队列以及处理中的元素（见图10.1左侧）。队列系统的基本法则是利特尔法则，该法则认为在稳定状态下，队列系统中的元素数

量（N）等于系统吞吐量（T）乘以总排队/服务时间（S），即N = T · S。另外两种形式是：T = N / S以及S = N / T，在计算中同样有用。

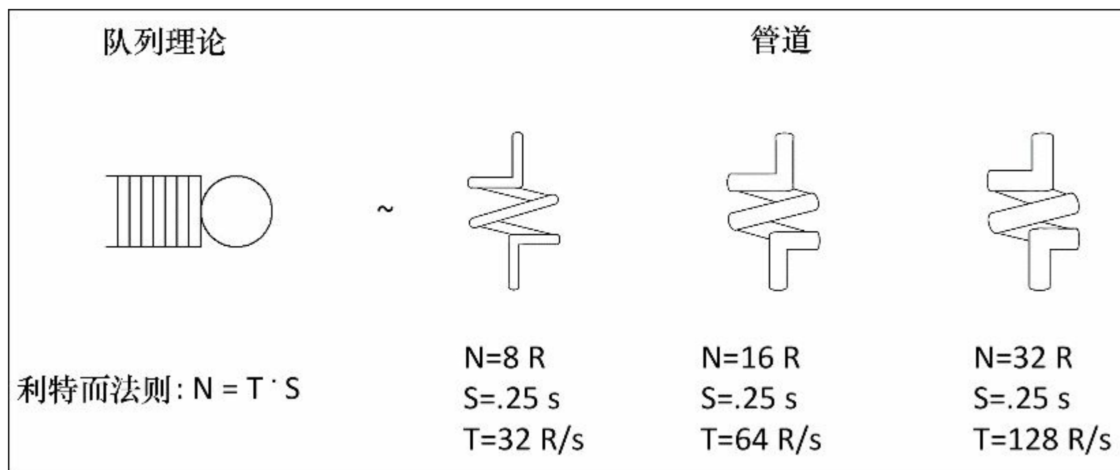


图10.1　利特尔法则、队列系统以及管道

在管道的几何形状中也有相似的法则（见图10.1右侧）。管道容量（V）等于管道长度L乘以横截面面积（A），即V = L·A。

如果我们想象长度表示服务时间（L～S），容量表示处理系统的元素数量（V～N），横截面面积表示吞吐量（A～N），那么利特尔法则和容量公式实际是相同的事情。



这个类比有道理吗？答案是差不多。如果我们将工作单位想象为小滴液体，以恒定速率在管道内部移动，那么L～S绝对有意义，因为管道越长，水滴移动花费的时间越多。V～N同样有意义，因为管道越大，能够容纳的水滴越多。烦人的是，我们还可以通过施加更大压力的方式压入更多水滴。A～T是不太满足类比的一点。在管道中，实际吞吐量，即每秒进出管道的水滴数量，被称为“体积流量”，除非满足特定条件（孔口），否则其与A^2成

> 正比，而不是A。这是因为更宽的管道不只意味着有更多的液体流出，还会使液体流动更快，因为管壁之间存在更大的空间。不过为了本章的学习，我们可以忽略这些技术细节，而是假设生活在一个理想的世界中，在这里压力和速度都是常量，并且吞吐量与横截面面积直接成正比。

利特尔法则和这个简单的体积公式非常相似，这就使得该“管道模型”非常直观有用。让我们更详细地看一下图10.1中的示例（右侧）。假设管道系统表示Scrapy的下载器。第一个非常“细”的下载器，其总体积/并发级别（N）可能是8个并发请求。管道长度/延迟（S）对于一个快速的网站来说，可能S=250ms。在给定N和S时，现在可以计算处理元素的体积/吞吐量，每秒请求数为T = N / S = 8 / 0.25 = 32。

你会发现延迟经常是我们无法控制的，因为它依赖于远端服务器的性能以及网络的延迟。我们比较容易控制的是下载器中并发（N）的级别，可以将其从8增长到16或32个并发请求，即10.1图中的第二个和第三个管道。对于常量的长度（超出我们控制范围之外），可以通过只增加横截面面积的方式增长体积，也就是说增加吞吐量！按照利特尔法则，16个并发请求时，我们得到的每秒请求数为T = N / S = 16 / 0.25 = 64个，而在32个并发请求时，我们得到的每秒请求数是T = N / S = 32 / 0.25 = 128个。太好了！我们似乎可以通过增加并发的方式，使系统无限快。在急于得出这样的结论之前，还需要考虑队列系统级联的影响。

10.1.1　级联队列系统

当将不同横截面面积/吞吐量的几个管道依次连接起来时，可以很直观地理解整个系统的流量将由最窄的（最小吞吐量：**T**）管道所限制（见图10.2）。

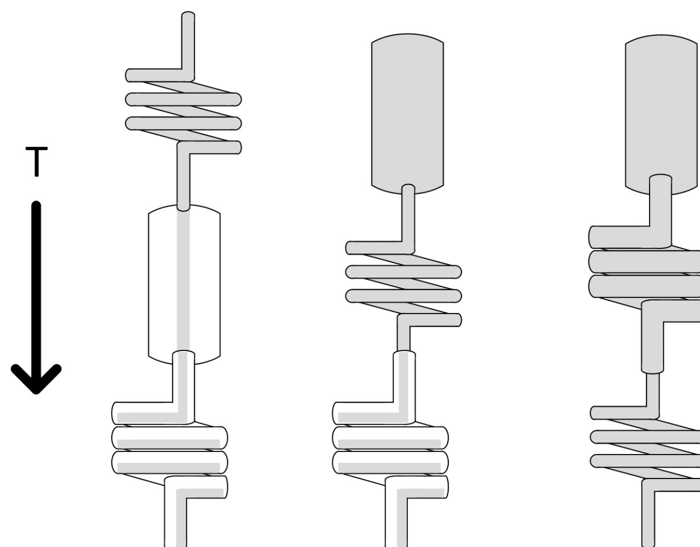


图10.2　不同容量的级联队列系统

你还可以观察到最窄管道（即瓶颈）的位置，决定了其他管道是如何“填满”的。如果考虑到与系统内存需求相关的填充，就会意识到瓶颈的位置是非常重要的。我们最好通过配置保持管道充满，且单个工作单元的花销最少。在Scrapy中，一个工作单元（爬取一个页面）主要是由下载器前的URL（几个字节）以及下载后的URL加上服务器响应（较大）组成。



这就是为什么在Scrapy系统中，通常将瓶颈放置在下载器中。

10.1.2　定义瓶颈

使用管道系统作为类比的一个非常重要的好处是，它在定义瓶颈的过程中更加直观。如果观察图10.2就会发现，“瓶颈”前的所有地方都是

满的，而之后的所有地方都不是。

好消息是，在大多数系统中，可以相对容易地使用系统度量监控队列系统是如何填满的。通过仔细检查Scrapy的队列，我们可以了解瓶颈在什么地方，如果发现不在下载器中，则可以调整设置让其变为下载器。没有改善瓶颈的任何改进都不会带来吞吐量的收益。如果修改系统其他部分，只会让事情变得更糟，很有可能将瓶颈转移到别的地方。这个感觉有点像追尾，可能需要很长时间，并且会令你感到绝望。你必须遵循系统方法，定义瓶颈，并且需要在修改任何代码或配置之前，“知道锤子应该击中哪里”。你在大部分例子中（包括本书的大多数例子）可以看到，瓶颈不是总在人们期望的地方出现。

10.1.3　Scrapy性能模型

让我们回到Scrapy，详细看一下其性能模型（见图10.3）。

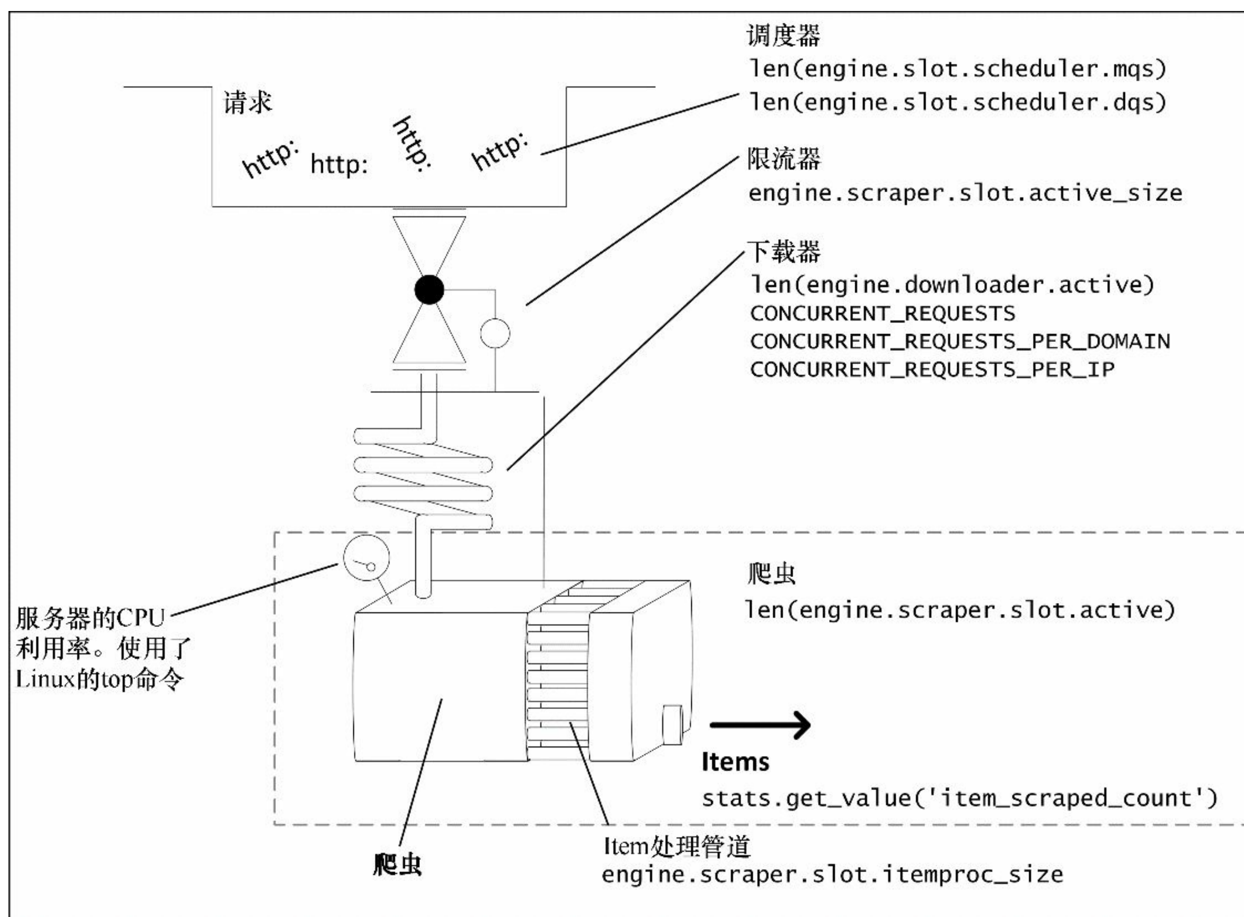


图10.3　Scrapy性能模型

Scrapy包含如下组成部分。

- 调度器： 在这里，多个请求会排队等待下载器处理。它们主要由URL组成，因此会十分紧凑，这就意味着即使拥有大量URL也不会对系统有很大伤害，并且可以让我们在传入不规则请求流的情况下能够充分利用下载器。
- 限流器： 这是抓取过程（大储水池）反馈的安全阀，如果正在执行的响应的总计大小超过5MB，那么它会让前往下载器的后续请求停止。这可能会导致不可预料的性能起伏。
- 下载器： 这是Scrapy关于性能最重要的组成部分。它对能够并行执

行的请求的数量有着复杂的限制。其延迟（管道长度）等于远程服务器响应的时间，加上所有网络/操作系统以及Python/Twisted的延迟。我们可以调整并行请求的数量，不过通常情况下，我们几乎无法控制延迟。下载器的容量由`CONCURRENT_REQUESTS*` 设置限制，我们将会很快看到。

- 爬虫： 这是抓取过程中将响应转为`Item` 和后续请求的部分。同时这也是我们编写的部分，通常情况下，只要遵照规则，它们就不会是性能瓶颈。
- **Item** 管道：这是我们编写的代码的第二个部分。我们的爬虫可以对每个请求生成上百个`Item` ，同一时刻只会处理`CONCURRENT_ITEMS` 个。该值十分重要，因为假设你在管道中要处理数据库访问，那么使用默认值（100）就可能会过高，从而在无意间拖垮数据库。

爬虫和管道都应该使用异步代码，并且在必要时引发更多的延迟，但不应因此成为瓶颈。极少情况下，我们的爬虫/管道会处理非常繁重的事情。如果发生此种情况，那么服务器的CPU可能会成为瓶颈。

10.2　使用telnet获得组件利用率

想要理解`Request/Item` 流是如何通过管道的，我们不会真得去测量流量（尽管这可能会是一个很棒的功能），而是使用更容易的方式测量Scrapy的每个处理阶段中存在多少流体，即`Request/Response/Item` 。

我们可以通过Scrapy运行的Telnet服务获取性能信息。首先，通过

使用telnet命令连接到6023 端口。然后，将会在Scrapy中得到一个Python提示符。需要小心的是，如果你在这里执行了某些阻塞操作，例如time.sleep()，它将会中止爬虫功能。内置的est()函数可以打印出一些感兴趣的度量。其中一些或者很专用，或者能够从几个核心度量推断出来。在本章剩余部分只会展示后者。让我们从一个示例运行中了解它们。当运行爬虫时，可以在开发机中打开第二个终端，通过telnet命令连接6023 端口，并运行est()。



本章代码位于ch10 目录，其中本例位于ch10/speed 目录。

在第一个终端中，运行如下代码。

```
$ pwd

/root/book/ch10/speed

$ ls

scrapy.cfg speed
```

```
$ scrapy crawl speed -s SPEED_PIPELINE_ASYNC_DELAY=1

INFO: Scrapy 1.0.3 started (bot: speed)
```

现在先不用管scrapy crawl speed是什么，以及其参数表示什么。本章后续部分会详细解释这些。现在，在第二个终端上，运行如下命令：

```
$ telnet localhost 6023

>>> est()

...
```

```
len(engine.downloader.active)      : 16

...

len(engine.slot.scheduler.mqs)     : 4475

...

len(engine.scraper.slot.active)    : 115

engine.scraper.slot.active_size    : 117760

engine.scraper.slot.itemproc_size   : 105
```

然后在第二个终端按下*Ctrl* + *D* 退出Telnet，回到第一个终端，按下*Ctrl* + *C* 停止爬虫。



我们在这里忽略了`dqs`。如果通过`JOBDIR`设置启用了持久化支持的话，还会得到非零的`dqs`（`len(engine.slot.scheduler.dqs)`），你需要将其添加到`mqs`的大小中，以继续后续分析。

我们来看一下本例中的这些核心度量都表示什么。`mqs`表示目前在调度器中还有很多等待（4475个请求）。还可以。`len(engine.downloader.active)`表示目前有16个请求正在下载器中被下载。这和我们在爬虫`CONCURRENT_REQUESTS`设置中设定的值相同，所以此处非常好。`len(engine.scraper.slot.active)`告知我们正在进行抓取处理的响应有115个。通过`(engine.scraper.slot.active_size)`，我们知道这些响应大小总计为115kb。在这些响应中，有 105 个Item此时正在通过管道处理，可以从`(engine.scraper.slot.itemproc_size)`看出来，这就意味着剩余的10个请求目前正在爬虫中处理。总体来说，我们可以看出瓶颈似乎在下载器中，在其之前的工作队列（`mqs`）非常庞大，但下载器已经满负荷利用了；而在其之后，我们有着数量很高但又比较稳定的任务（可以通过多次执行`est()`来确认此项）。

我们感兴趣的另一个信息元是stats 对象，即通常在爬取完成后打印的信息。我们可以在Telnet中，通过stats.get_stats()，以字典的形式在任何时间访问它，并且可以通过p() 函数打印更优雅的格式。

```
$ p(stats.get_stats())

{'downloader/request_bytes': 558330,

...

 'item_scraped_count': 2485,

...}
```

对我们来说，目前最感兴趣的度量是item_scraped_count，它可

以通过stats.get_value('item_scraped_count') 直接访问。该度量告知我们到目前为止有多少item 已经被抓取，它应当以系统吞吐量（Item /秒）的速率增长。

10.3 基准系统

为了第10章，我编写了一个简单的基准系统，可以让我们在不同场景下评估性能。该系统的代码比较复杂，你可以在speed/spiders/speed.py 中找到它，但我不会详细讲解该代码。

该系统包含如下功能。

- 我们的Web服务器上http://localhost:9312/benchmark/... 目录的处理器。可以通过调整URL参数/Scrapy设置控制伪站点的结构（见图10.4）以及页面加载速度。无需担心细节，我们很快就会看到更多示例。现在，可以观察 http://localhost:9312/benchmark/index?p=1 与http://localhost: 9312/benchmark/ id:3/rr:5/index?p=1 的区别。第一个页面加载时间在半秒之内，并且每个详情页中有一个条目；而第二个页面需要5秒时间加载，但每个详情页中包含3个条目。我们还可以向页面中添加一些隐藏的垃圾数据，使其更大一些。比如，http://localhost:9312/ benchmark/ds:100/ detail?id0=0 。默认情况下（参见speed/settings.py ），页面渲染在SPEED_T_RESPONSE = 0.125 秒内，伪站点包含SPEED_TOTAL_ITEMS = 5000 个Item 。

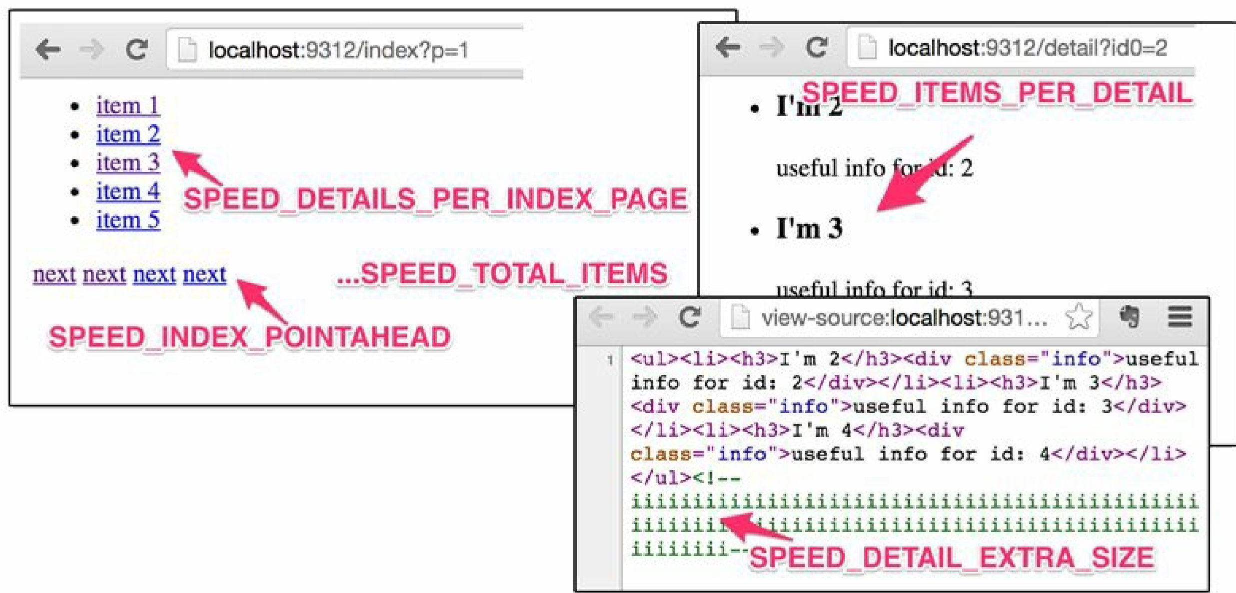


图10.4　我们的基准系统创建的具有可调整结构的伪站点

- 爬虫SpeedSpider，通过控制SPEED_START_REQUESTS_STYLE 设置伪造一些获取start_requests() 的方式，并提供了一个简单的parse_item() 方法。默认情况下，我们使用crawler.engine.crawl() 方法直接将所有启动URL提供给Scrapy的调度器。
- 管道DummyPipeline 伪造一些处理。它包含该处理可能导致的4种延迟类型：阻塞/计算/同步延迟（SPEED_PIPELINE_BLOCKING_DELAY，这是一种不好的方式）、异步延迟（SPEED_PIPELINE_ASYNC_DELAY，这是一种可以接受的方式）、使用treq 库的远程API调用（SPEED_PIPELINE_API_VIA_TREQ，这是一种可以接受的方式）以及使用Scrapy的crawler.engine.download() 的远程API调用（SPEED_PIPELINE_API_VIA_DOWNLOADER，这是一种不太好的方式）。默认情况下，该管道不会添加任何延迟。

- 在`settings.py`中包含了一组高性能设置。所有可能会造成系统有任何减慢的设置都已经被禁用。由于我们只访问本地服务器，因此针对单域名请求数的限制也被禁用了。
- 与第8章类似的少量度量捕获扩展。它将周期性地打印出核心度量指标。

我们已经在前面的例子中使用了该系统，不过让我们重新运行一次模拟，并使用Linux的时间工具测量完整的执行时间。可以在如下代码中看到被打印出来的核心度量指标。

```
$ time scrapy crawl speed

...

INFO: s/edule  d/load  scrape  p/line   done    mem

INFO:       0       0       0       0      0      0

INFO:    4938      14      16       0     32  16384
```

```
INFO:    4831      16       6       0    147   6144

...

INFO:     119      16      16       0   4849  16384

INFO:       2      16      12       0   4970  12288

...

real 0m46.561s
```

Column	Metric

s/edule	len(engine.slot.scheduler.mqs)
d/load	len(engine.downloader.active)
scrape	len(engine.scraper.slot.active)
p/line	engine.scraper.slot.itemproc_size
done	stats.get_value('item_scraped_count')
mem	engine.scraper.slot.active_size

这种级别的透明度是非常明显的。我缩短了列名，不过它们应该仍然能够清楚说明含义。初始时，在调度器中有5000个URL，而在结束时，完成列中也有5000个item。下载器作为瓶颈，已经被充分利用，根据设置始终会有16个活跃的请求。抓取操作主要是爬虫，因为如我们在`p/line`列所见，管道是空的，由于它通常是在瓶颈之后，因此虽然一定程度上被利用了，但是没有充分利用。抓取5000个`Item`花费了46秒的时间，使用的并发请求N = 16，即每个请求的平均时间是46 · 16 / 5000 = 147ms，而不是我们期望的125ms，不过这也还可以接受。

10.4　标准性能模型

标准性能模型在Scrapy功能正常且下载器为性能瓶颈时成立。在这种情况下，可以在调度器中看到一些请求，而在下载器中则是并发请求

数的最大值（见图10.5）。抓取程序（爬虫和管道）被轻度加载，并且处理中的响应数不会持续增长。

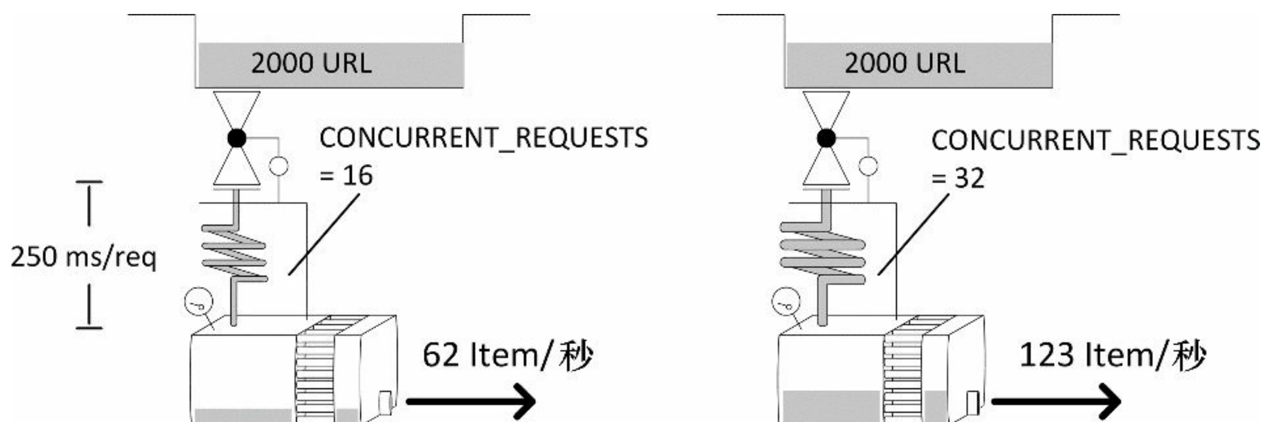


图10.5　标准性能模型及一些实验结果

有 3 个主要设置用于控制下载器能力：CONCURRENT_REQUESTS、CONCURRENT_REQUESTS_PER_DOMAIN 以及 CONCURRENT_REQUESTS_PER_IP 。其中第一个是粗调控制。无论如何都不会在同一时间有超过CONCURRENT_REQUESTS 数量的请求处于活跃状态。而如果你的目标是单个域名或相对较少的几个域名，CONCURRENT_REQUESTS_PER_DOMAIN 可能会进一步限制活跃请求的数量。如果设置了CONCURRENT_REQUESTS_PER_IP ，那么CONCURRENT_REQUESTS_PER_DOMAIN 就会被忽略，此时有效的限制将会是针对单个（目标）IP的请求数。比如，当目标是一些共享主机站点时，多个域名可能会指向同一台服务器，该设置可以帮助你不会过度攻击该服务器。

为了保持现在的性能探索尽可能简单，我们通过使CONCURRENT_REQUESTS_PER_IP 保留为默认值（0）以禁用每个IP的限制，并且设置CONCURRENT_REQUESTS_PER_DOMAIN 的值为非常大的

数值（1000000）。这样的组合可以有效禁用针对IP和域名的限制，下载器的并发数量可以完全由`CONCURRENT_REQUESTS`来控制。

我们希望系统吞吐量依赖于下载页面所花费的平均时间，包括远程服务器部分以及我们的系统（Linux、Twisted/Python）的延迟（$t_{download} = t_{response} + t_{overhead}$）。如果能够考虑一些启动和结束时间也是很好的。它包括你得到一个响应的时间与其Item从管道另一端出来的时间之间的间隔，以及在缓存冷启动时，你得到第一个响应之前的时间及性能较差时的时间。

总之，如果你需要完成N个请求的任务，并且我们的爬虫已经得到了适当的调整，那么你应该会在下述公式所得的时间内完成。

$$t_{job}\frac{\mathrm{N}\cdot(t_{response}+t_{overhead})}{\mathrm{CONCURRENT_REQUESTS}}+t_{start/stop}$$

我们无法控制这些参数中的大部分，这多少让人有些遗憾。我们可以使用一台更强大的服务器来稍微控制$t_{overhead}$，类似情况还有t_{start}/t_{stop}（该参数几乎不值得为之努力，因为我们只会在每次运行时才会花费该时间）。除了对N个请求的给定工作量有少许改善外，我们所能细心调整的数值只有`CONCURRENT_REQUESTS`，它通常依赖于我们访问远程服务器的困难程度。如果我们将其设定为一个非常大的数值，在某一时刻，会使服务器的CPU能力或远程服务器及时响应的能力达到饱和，也就是说，$t_{response}$将会突增，因为目标网站对我们实施了限速、封禁，或者我们造成了目标网站宕机。

让我们运行一个实验来检查我们的理论。我们将以$t_{response} \in \{0.125s, 0.25s, 05s\}$、`CONCURRENT_REQUESTS` $\in \{8, 16, 32, 64\}$的条件爬

取2000个item，如下所示。

```
$ for delay in 0.125 0.25 0.50; do for concurrent in 8 16 32 64; do

    time scrapy crawl speed -s SPEED_TOTAL_ITEMS=2000 \

    -s CONCURRENT_REQUESTS=$concurrent -s SPEED_T_RESPONSE=$delay

done; done
```

在我的笔记本上，完成2000个请求的时间如表10.1所示（以秒为单位）。

表**10.1**

CONCURRENT_REQUESTS	**125ms/**请求	**250ms/**请求	**500ms/**请求
8	36.1	67.3	129.7

16	19.4	35.3	66.1
32	11.1	19.3	34.7
64	7.4	11.1	19.0

警告： 接下来将会是令人讨厌的计算！你可以略读本段内容。我们可以在图10.5中看到部分结果。通过重新排列最后的公式，我们可以将其转换为更加简单的形式（即$y = t_{overhead} \cdot x + t_{start/stop}$，其中$x$ = N / CONCURRENT_REQUESTS和$y = t_{job} \cdot x + t_{response}$）。使用最小二乘法（Excel函数为`LINEST`）和前面的数据，我们可以计算得到$t_{overhead}$ = 6ms，而$t_{start/stop}$ = 3.1s。$t_{overhead}$是一个很小的数值，而启动时间却非常显著，不过它支持了数千个URL的长时间运行。因此，我们将使用一个非常有用的公式，以请求数/秒为单位近似系统的吞吐量，如下所示。

$$T = \frac{\mathrm{N}}{t_{job} - t_{start/stop}}$$

通过运行N个请求的长时间任务，我们可以测量出t_{job}的汇总时间，然后直接计算T。

10.5　解决性能问题

现在我们应当对系统预期拥有的性能是什么有了充分的了解，接下来看一下如果没有得到想要的性能时应当如何操作。我们将通过探讨具体症状来展示不同的问题案例，执行示例爬虫进行复现，探讨根本原

因，最终提供解决问题的操作。案例展示的顺序从系统顶层问题逐步到低层次的Scrapy技术细节。这就意味着更普遍的案例可能会出现在没那么常见的案例之后。在探索你的性能问题之前，请完整阅读本章全部内容。

10.5.1　案例 #1：CPU饱和

症状：在某些情况下，你增加了并发级别，但没有得到性能提升。当降低并发级别时，一切工作再次回归预期（见图10.6）。你的下载器可以被充分利用，但是似乎每个请求的平均时间出现了激增。当在UNIX/Linux系统中使用`top`命令、在Power Shell中使用`ps`命令或在Windows中使用任务管理器查看CPU负载如何时，会发现CPU负载非常高。

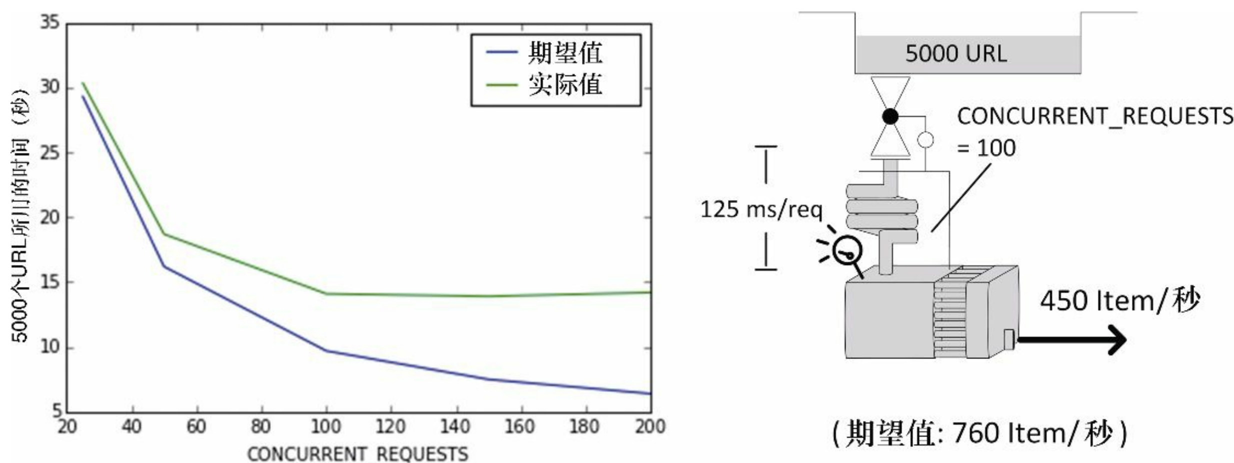


图10.6　当并发增长到一定程度后，性能趋于平缓

示例：假设运行了如下命令。

```
$ for concurrent in 25 50 100 150 200; do
```

```
  time scrapy crawl speed -s SPEED_TOTAL_ITEMS=5000 \
    -s CONCURRENT_REQUESTS=$concurrent
done
```

你得到了其抓取5000个URL的时间。在表10.2中，期望值一列是基于前面得到的公式计算所得，而CPU负载是通过`top` 命令观察得到的（可以在开发机中使用第二个终端运行该命令）。

表**10.2**

CONCURRENT_REQUESTS	期望值（秒）	实际值（秒）	期望值与实际值的百分比	**CPU**负载
25	29.3	30.34	97%	52%
50	16.2	18.7	87%	78%

100	9.7	14.1	69%	92%
150	7.5	13.9	54%	100%
200	6.4	14.2	45%	100%

在我们的实验中，由于几乎不执行任何处理，因此能够得到高并发。而在一个更复杂的系统中，很可能会更早地看到该行为。

讨论： Scrapy重度使用单一线程，当达到很高级别的并发时，CPU可能会成为瓶颈。假设不使用任何线程池，那么Scrapy应当使用的CPU负载建议在80%～90%。请记住你可能在其他系统资源上遇到相似的问题，比如网络带宽、内存或磁盘吞吐量，不过这些都很少见，并且会落入通用系统的管理范畴，因此就不在这里进一步强调了。

解决方案： 通常假设你的代码是有效的。你可以通过在同一台服务器上运行多个Scrapy爬虫，以使总计并发超过CONCURRENT_REQUESTS。这可以帮助你利用更多可用核心，尤其是当管道的其他服务或其他线程不使用它们的时候。如果需要更多的并发，可以使用多台服务器（参见第11章），这种情况下可能还需要更多可用的资金、网络带宽以及磁盘吞吐量。始终检查CPU利用率是你的首要约束。

10.5.2　案例 #2：代码阻塞

症状： 你所观察到的行为无法说通。和期望值相比，系统非常

慢，并且奇怪的是，即使当你改变CONCURRENT_REQUESTS 的值时，速度也没有显著变化（见图10.7）。下载器看起来总是空的（少于CONCURRENT_REQUESTS ），而抓取程序却有不少响应。

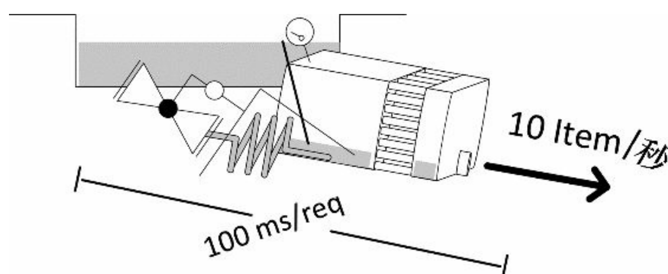


图10.7　阻塞代码以不可预测的方式使并发无效

示例： 你可以使用两个基准设置：SPEED_SPIDER_BLOCKING_DELAY 和SPEED_PIPELINE_BLOCKING_DELAY （它们具有相同的效果），对每个响应启用一个100ms的阻塞。在给定并发级别时，我们期望100个URL应当花费2～3秒，但无论CONCURRENT_REQUESTS 的值是多少，我们总是需要花费大约13秒的时间（见表10.3）。

```
for concurrent in 16 32 64; do

  time scrapy crawl speed -s SPEED_TOTAL_ITEMS=100 \

  -s CONCURRENT_REQUESTS=$concurrent -s SPEED_SPIDER_BLOCKING_DELAY=0.1
```

```
done
```

表**10.3**

CONCURRENT_REQUESTS	总时间（秒）
16	13.9
32	13.2
64	12.9

讨论： 任何阻塞代码都会立即抵消掉Scrapy的并发性，本质上相当于设置`CONCURRENT_REQUESTS` = 1。根据上面的简单公式，100URL · 100ms（阻塞延迟）= 10秒 + $t_{start/stop}$， 充分解释了我们所看到的延迟。

无论阻塞代码是在管道中还是在爬虫中，你都会发现抓取程序可以被充分利用，但其前后的模块都是空的。这看起来违背了前面讲过的管

道的物理现象，不过由于我们已经不再拥有一个并发系统了，所以管道规则不再适用。该错误非常容易发生（比如使用阻塞API），你一定会在某一时刻出现该错误。你会注意到类似的讨论同样适用于复杂代码的计算。你应当为此类代码使用多线程，正如我们在第9章中所看到的；或者是在Scrapy之外进行批量处理，我们将会在第11章中看到一个相关示例。

解决方案：将假设你继承了基代码，并且不清楚阻塞代码位于何处。如果该系统在没有任何管道的情况下仍然可以工作，那么禁用这些管道，并检查是否仍存在奇怪的行为。如果仍存在，那么阻塞代码位于爬虫中。如果不再存在，那么依次启用管道，观察问题是否开始出现。如果该系统在缺少任何运行中的模块的情况下无法正常运转，那么可以在每个管道阶段的功能之间添加一些日志消息（或插入虚拟管道打印时间戳）。通过检查日志，可以轻松检测出系统在什么地方花费了最多的时间。如果希望有一个更加长期/可复用的解决方案，可以使用虚拟管道跟踪你的请求，在`Request`的`meta`字段中为每个阶段添加时间戳。最后，hook到`item_scraped`信号，并记录时间戳日志。一旦你发现阻塞代码，则应将其转换为Twisted/异步代码，或使用Twisted的线程池。如果想要查看该转换的效果，可以将`SPEED_PIPELINE_BLOCKING_DELAY`替换为`SPEED_PIPELINE_ASYNC_DELAY`，重新运行前面的示例。性能的变化将十分惊人。

10.5.3　案例 #3：下载器中的“垃圾”

症状：你得到的吞吐量低于预期。下载器看起来有时会有比

CONCURRENT_REQUESTS 更多的请求。

示例： 模拟以0.25秒响应时间的情况下载1000个页面。按照默认的16个并发，根据公式需要花费大约19秒的时间。我们使用一个管道，用crawler.engine.download()制造到伪造API的额外HTTP请求，其响应时间在 1 秒之内。你可以通过http://localhost:9312/benchmark/ar:1/api?text=hello进行尝试（见图10.8）。让我们运行一个爬取程序。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_

RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_

DOWNLOADER=1

...

s/edule  d/load  scrape  p/line  done    mem

   968      32      32      32     0  32768
```

```
    952      16       0       0    32      0

    936      32      32      32    32  32768

...

real 0m55.151s
```

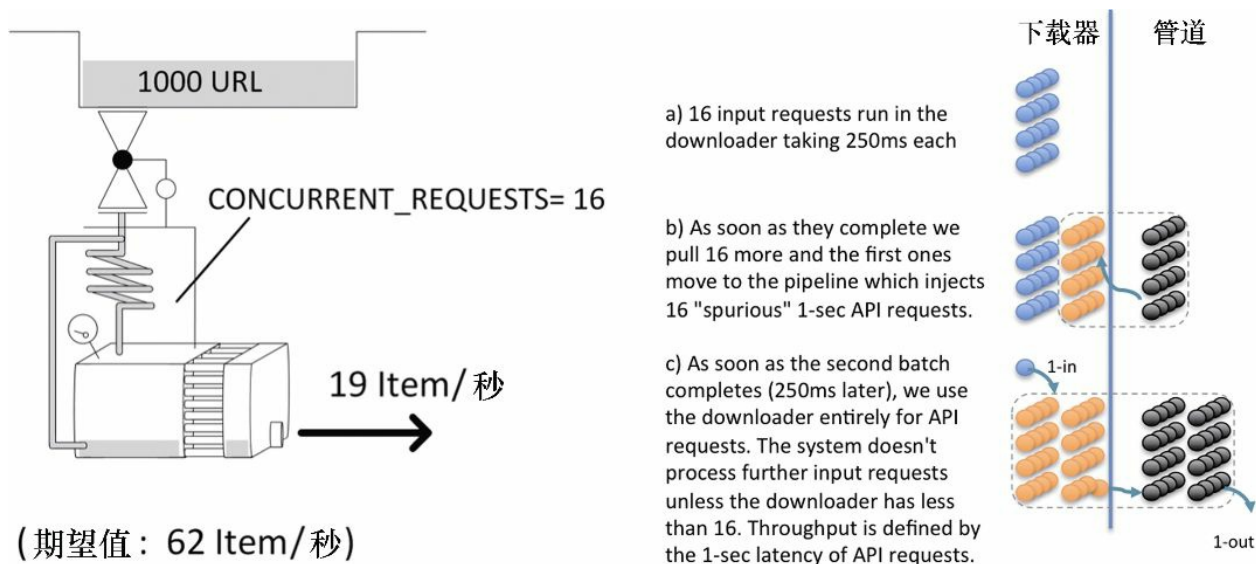


图10.8　由虚假API请求数定义的性能

非常奇怪！我们的任务不但花费了预期的3倍时间，还超出了下载器定义的`CONCURRENT_REQUESTS`所设定的16个活跃请求数（`d/load`）。下载器显然是瓶颈，因为它在超负荷工作。我们重新运行爬取程序，并在另一个控制台中打开到Scrapy的telnet连接。之后，就可以查看下载器中有哪些请求是活跃的了。

```
$ telnet localhost 6023

>>> engine.downloader.active

set([<POST http://web:9312/ar:1/ti:1000/rr:0.25/benchmark/api>, ... ])
```

看起来它处理的大部分是API请求，而不是下载正常页面。

讨论： 你可能会认为没有人使用`crawler.engine.download()`，因为它看起来会比较复杂，不过它在Scrapy的基代码中使用了两次，分别是`robots.txt`中间件和多媒体管道。因此，当人们需要使用Web API时，它也会被推荐为一种解决方案。因为使用它要比使用阻塞API更好，比如我们在前面章节中看到的流行的Python包`requests`；而且，使用它还会比理解Twisted编程和使用`treq`简单一些。现在既然有了咱们这本书，这些就不再是使用它的借口了。另一方面，该错误非常难调试，所以应当在研究性能时主动检查下载器中的活跃请求。如果发现API或多媒体URL不是你爬取的直接目标，那么就意味着某些管道使用了`crawler.engine.download()`来执行HTTP请求。由于我们的`CONCURRENT_REQUESTS`限制不适用于这些请求，也就意味着我们很可能看到下载器加载的请求数超过`CONCURRENT_ REQUESTS`，乍看起来有些矛盾。除非虚假请求数降低到`CONCURRENT_ REQUESTS`以下，否则调度器不会获取新的正常页面请求。

因此，我们从系统中得到的吞吐量相当于原始请求持续1秒（API延迟），而不是0.25秒（页面下载延迟）的吞吐量不是一种巧合。这种情况特别容易令人困惑，因为除非API调用比页面请求慢，否则我们不会注意到任何性能下降。

解决方案：我们可以使用`treq`代替`crawler.engine.download()`来解决该问题。你将发现这会使抓取程序的性能突增，这对于API架构来说可能是个坏消息。我将从一个低数值的CONCURRENT_REQUESTS开始，逐渐增长以确保不会使API服务器过载。

下面是和前面相同的运行示例，不过这次使用了`treq`。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_

RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_TREQ=1

...

s/edule  d/load  scrape  p/line  done    mem

    936      16      48      32     0  49152

    887      16      65      64    32  66560
```

```
    823      16      65      52    96  66560

...

real 0m19.922s
```

你会发现一个非常有趣的事情。管道（p/line ）似乎包含比下载器（d/load ）更多的条目（见图10.9）。这种情况非常好，并且了解其原因也很有趣。

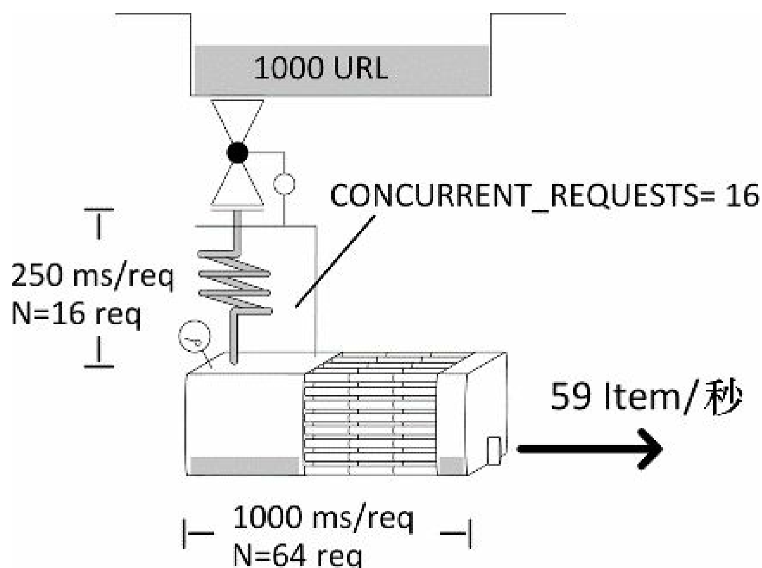


图10.9　拥有长管道非常完美（在Google图片中查看“industrial heat exchanger”）

下载器如预期一样，充分加载了16个请求。也就是说，系统吞吐量为T = N / S = 16 / 0.25= 64个请求/秒。我们可以通过观察done 列的增长进行确认。一个请求会在下载器中花费0.25秒，但是由于缓慢的API请求，它会在管道中花费1秒的时间。这意味着在管道中（p/line），我们期望看到平均N = T·S = 64·1 = 64个Item。非常好。这表示现在管道有瓶颈吗？不，因为我们没有限制同时在管道中处理的响应数量。只要数值不是无限增加，就能够很好地运行。在下一节中，我们将看到更多关于这个问题的讨论。

10.5.4　案例 #4：大量响应或超长响应造成的溢出

症状：下载器几乎满负荷运转，并且一段时间后关闭。该模式不断重复。抓取程序的内存使用率很高。

示例：此处我们使用了和前面一样的设置（使用了treq ），不过响应会比较大，大约是120KB的HTML。如你所见，此时花费了31秒的

时间完成，而不是20秒左右（见图10.10）。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=1000 -s SPEED_T_

RESPONSE=0.25 -s SPEED_API_T_RESPONSE=1 -s SPEED_PIPELINE_API_VIA_TREQ=1

-s SPEED_DETAIL_EXTRA_SIZE=120000

s/edule  d/load  scrape  p/line  done     mem

   952      16      32      32     0  3842818

   917      16      35      35    32  4203080

   876      16      41      41    67  4923608
```

```
    840       4      48      43   108  5764224

    805       3      46      27   149  5524048

...

real 0m30.611s
```

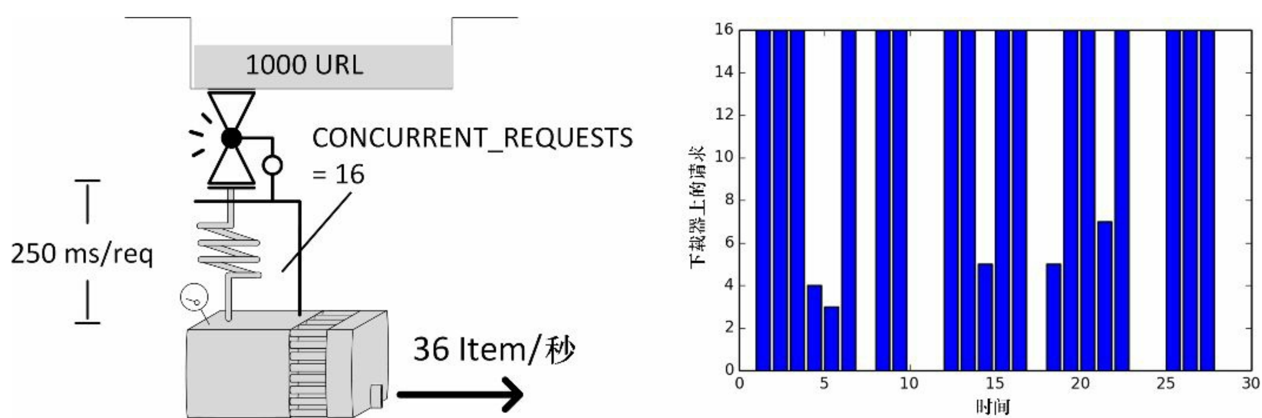


图10.10　下载器中不规则的请求数表示响应大小限流

讨论：我们可能会天真地尝试将这种延迟解释为“创建、传输、处

理页面需要花费更多时间”，不过这并不是此处发生的情况。此处有一个硬编码（编写代码时写入）的对请求总大小的限制：`max_active_size` = 5000000。假设每个请求的大小等于其请求体的大小，并且至少是1KB。

一个重要的细节是，该限制可能是Scrapy最巧妙且本质的机制，用于防止过慢的爬虫或管道。如果你的任何一个管道的吞吐量比下载器的吞吐量更慢，最终就会发生这种情况。当管道处理时间过长时，即使很小的请求，也很容易触发该限制。下面是一个管道超长的极端案例，80秒之后就会开始产生问题。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=10000 -s SPEED_T_

RESPONSE=0.25 -s SPEED_PIPELINE_ASYNC_DELAY=85
```

解决方案： 对于已存在的基础架构，针对该问题几乎无计可施。当你不再需要时（比如爬虫之后），清空响应体是个不错的选择，不过在写操作时执行该操作不会重置Scraper的计数器。所有你能做的就是降低管道的处理时间，从而有效减少Scraper中处理的响应数量。可以使用

传统的优化手段实现它：检查可能与之交互的API或数据库是否能够支持抓取程序的吞吐量；分析抓取程序；将功能管道移动到批处理/后处理系统；使用更强大的服务器或分布式爬取。

10.5.5　案例 #5：有限/过度item并发造成的溢出

症状： 你的爬虫为每个响应创建了多个Item。你得到的吞吐量低于预期，并且可能和前面案例中的开/关模式相同。

示例： 这里，我们有一个稍微不太一样的设置，我们有1000个请求，并且它们的每个返回页面都有100个Item。响应时间是0.25秒，Item管道处理时间为3秒。我们设置CONCURRENT_ITEMS 的值从10到150，执行多次。

```
for concurrent_items in 10 20 50 100 150; do

time scrapy crawl speed -s SPEED_TOTAL_ITEMS=100000 -s \

SPEED_T_RESPONSE=0.25 -s SPEED_ITEMS_PER_DETAIL=100 -s \

SPEED_PIPELINE_ASYNC_DELAY=3 -s \
```

```
CONCURRENT_ITEMS=$concurrent_items

done

...

s/edule  d/load  scrape  p/line  done     mem

    952      16      32     180     0  243714

    920      16      64     640     0  487426

    888      16      96     960     0  731138

...
```

讨论： 值得再次注意，该情况只适用于爬虫为每个响应生成多个Item时。除这种情况外，你应该设置`CONCURRENT_ITEMS = 1`，然后忘了它。另外还需注意的是，这是一个虚拟的示例，因为其吞吐量相当大，达到了每秒大约1300个Item。之所以达到如此高的吞吐量，是因为延迟低且稳定、几乎没有真实处理，以及响应的大小很小。这种情况并不常见。

我们首先要注意的事情是，在此之前`scrape`和`p/line`列通常都是相同的数值，而现在`p/line`则是`CONCURRENT_ITEMS · scrape`。这是符合预期的，因为`scrape`显示的是响应数，而`p/line`则是`Item`数。

第二个有意思的事情是图10.11所示的浴缸形状的性能函数。由于纵轴是缩放的，因此该图表看起来会比实际情况更显著。在左侧，延迟非常高，因为触及了前一节所提到的内存限制。而在右侧，并发过多，造成使用了过多的CPU。获得最佳效果并不那么重要，因为向左右移动非常容易。

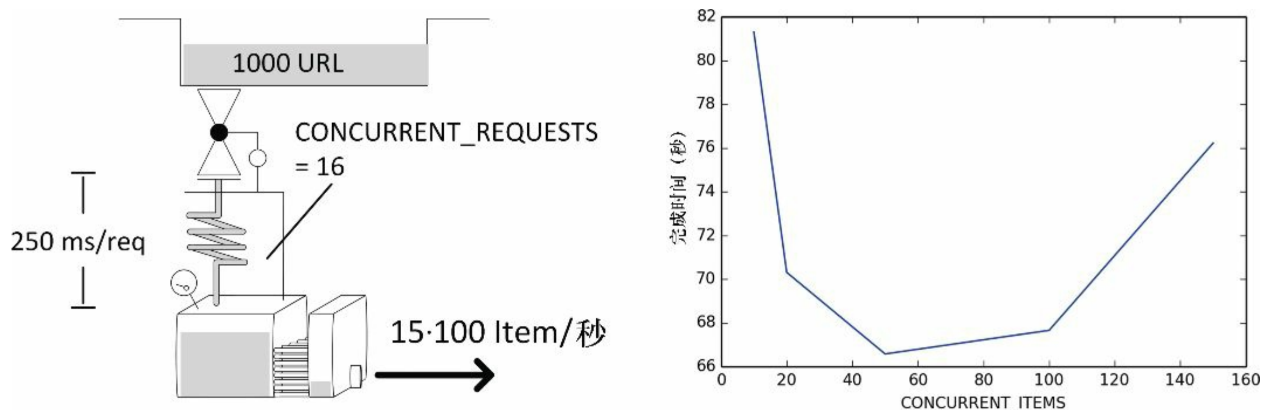


图10.11　以CONCURRENT_ITEMS为变量的爬取时间函数

解决方案：检测本案例的两种问题症状非常容易。如果CPU使用率过高，那么最好减少CONCURRENT_ITEMS 的值。如果触及响应的5MB限制，那么你的管道无法跟上下载器的吞吐量，增加CONCURRENT_ITEMS 的值可能能够快速解决该问题。如果修改后没有什么区别，那么应当遵照前面一节给出的建议，再三询问自己系统的其余部分是否能够支持你的抓取程序的吞吐量。

10.5.6　案例 #6：下载器未充分利用

症状：你增加了CONCURRENT_REQUESTS 的值，但是下载器并未跟上，没能充分利用。调度器是空的。

示例：首先，我们运行一个没有问题的示例。我们将切换到1秒的响应时间，因为它能够简化计算量，使下载器的吞吐量T = N / S = N / 1 = CONCURRENT_REQUESTS 。假设按照如下命令运行。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 \
```

```
-s SPEED_T_RESPONSE=1 -s CONCURRENT_REQUESTS=64

s/edule  d/load  scrape  p/line  done  mem

    436     64       0       0     0    0

...

real 0m10.99s
```

我们得到了一个充分利用的下载器（64个请求），总时间为11秒，与我们以每秒64个请求的条件处理500个URL的模型相匹配（$S = N / T + t_{start/stop} = 500 / 64 + 3.1 = 10.91$秒）。

现在，执行相同的爬取，不过不再像前面那些示例那样默认从列表

中提供URL，而是使用索引页通过`SPEED_START_REQUESTS_STYLE=UseIndex`抽取URL。这和我们本书中其他章使用的模式相同。每个索引页默认包含20个URL。

```
$ time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 \

-s SPEED_T_RESPONSE=1 -s CONCURRENT_REQUESTS=64 \

-s SPEED_START_REQUESTS_STYLE=UseIndex

s/edule  d/load  scrape  p/line  done   mem

      0       1       0       0     0     0

      0      21       0       0     0     0

      0      21       0       0    20     0
```

```
...

real 0m32.24s
```

很明显，这和前面的案例不太一样。不知为何，下载器的运行低于其最大能力，并且吞吐量为$T = N / S - t_{start/stop}$ = 500 / (32.2 - 3.1) = 17个请求/秒。

讨论：快速浏览`d/load`列，可以确信下载器没能充分利用。这是因为我们没有足够的URL提供给它。我们的抓取处理生成URL的速度比最大消费能力要慢。在本例中，每个索引页会生成20个URL加上1个前往下一索引页的URL。吞吐量无论如何都无法超过每秒20个请求，因为我们无法足够快地得到源URL。该问题非常隐蔽，容易被忽视。

解决方案：如果每个索引页包含一个以上的下一页的链接，那么可以利用它们加速URL的生成。如果可以找到显示更多结果的索引页面（比如50个）就更好了。我们可以通过运行几个模拟来观察其行为。

```
$ for details in 10 20 30 40; do for nxtlinks in 1 2 3 4; do
```

```
time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 -s SPEED_T_RESPONSE=1 \

-s CONCURRENT_REQUESTS=64 -s SPEED_START_REQUESTS_STYLE=UseIndex \

-s SPEED_DETAILS_PER_INDEX_PAGE=$details \

-s SPEED_INDEX_POINTAHEAD=$nxtlinks

done; done
```

在图10.12中，可以看到吞吐量是如何根据这两个参数变化的。我们观察到了线性行为，无论是下一页链接，还是详情页，直到达到系统上限。可以通过重新排列爬取的`Rule`进行实验。如果使用LIFO（默

认）顺序，你可能会看到如果先调用索引页请求，最后在列表中抽取它们的话，能够得到较小的改善。你也可以尝试为访问索引页的请求设置高优先级。虽然这两种技术都没有显著的改善，但可以通过分别设置`SPEED_INDEX_RULE_LAST=1`和`SPEED_INDEX_HIGHER_PRIORITY=1`来进行尝试。请注意这两种解决方案都会首先下载整个索引页（由于优先级高），因此会在调度器中生成大量URL，增加内存需求。在它们完成所有索引之前，只会给出少量的结果。对于少量索引还可以接受，但是对于大量索引的情况，就不太可取了。

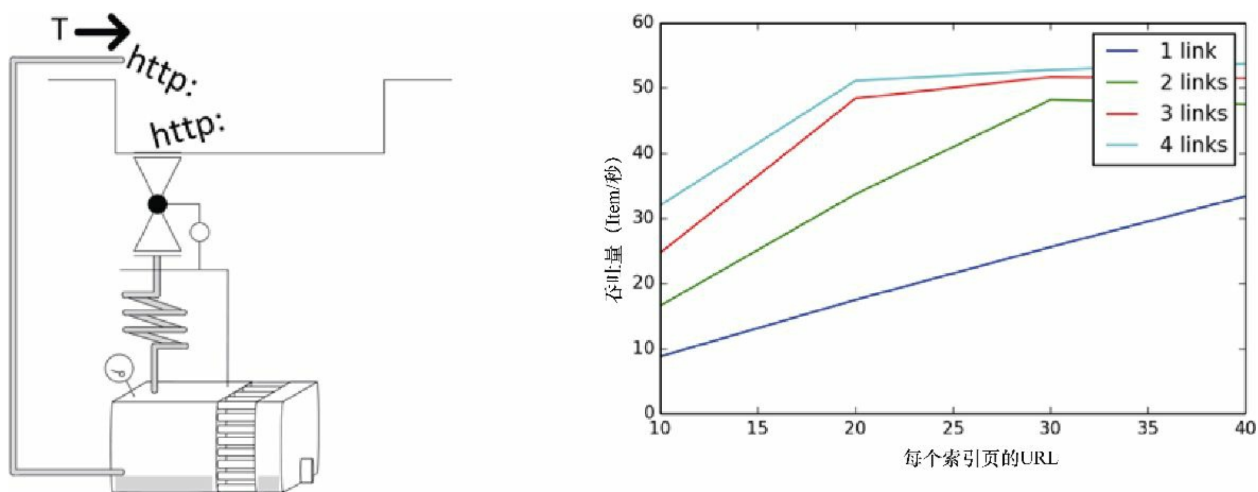


图10.12　以每个索引页链接的详情页及下一页数量为变量的吞吐量函数

一个简单而又强大的技术是索引分片。这就需要你使用超过一个初始索引URL，在它们之间有一个最大距离。比如，如果索引包含100页，你可以选取1和51作为起始索引。然后，爬虫可以以两倍速率使用下一页链接有效遍历索引。如果你能找到一种遍历索引的方式，比如基于产品的品牌或提供给你的任何其他属性，并且可以将其按照大致相等的段进行拆分的话，也可以做到类似的事情。你可以使用`-s SPEED_INDEX_SHARDS`设置进行模拟。

```
$ for details in 10 20 30 40; do for shards in 1 2 3 4; do
time scrapy crawl speed -s SPEED_TOTAL_ITEMS=500 -s SPEED_T_RESPONSE=1 \
-s CONCURRENT_REQUESTS=64 -s SPEED_START_REQUESTS_STYLE=UseIndex \
-s SPEED_DETAILS_PER_INDEX_PAGE=$details -s SPEED_INDEX_SHARDS=$shards
done; done
```

结果要比前面的技术更好，如果该方法适合你的话，我将会推荐这种方法，因为它更加简单整洁。

10.6　故障排除流程

总结来说，Scrapy在设计时就将下载器作为瓶颈。从一个低数值的`CONCURRENT_REQUESTS`开始，逐渐增加，直到触及下述限制之一：

- CPU使用率大于80%～90%；
- 源网站延迟过度增长；
- 抓取程序中响应达到了5MB的内存限制。

同时，执行以下操作：

- 始终保持调度器队列（mqs/dqs）中至少有一定量的请求，避免下载器出现URL饥饿；
- 永远不要使用任何阻塞代码或CPU密集型代码。

图10.13总结了诊断并修复Scrapy性能问题的过程。

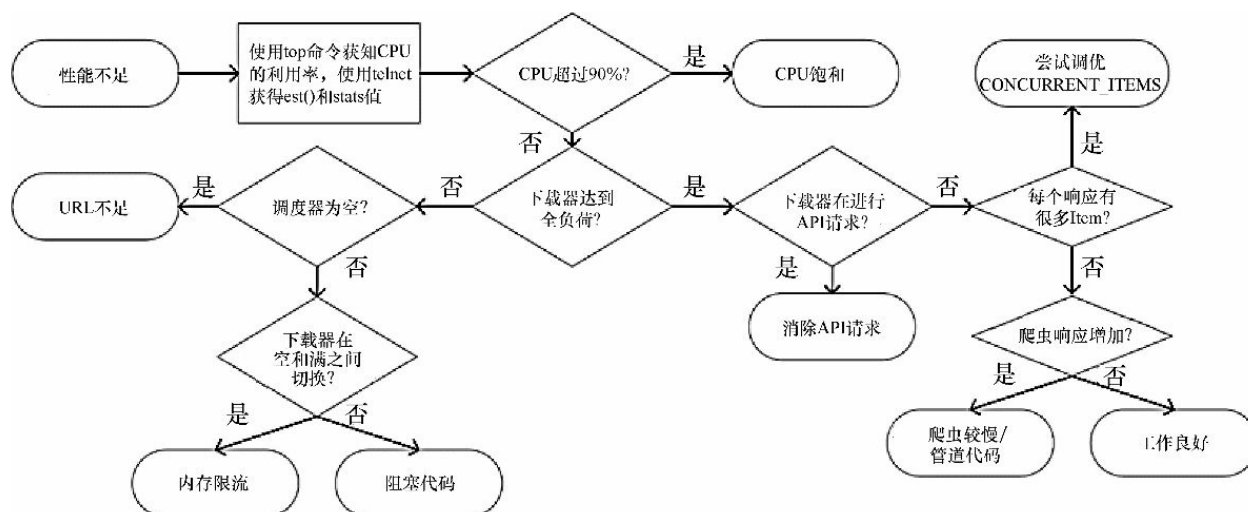


图10.13　Scrapy性能问题故障排除

10.7　本章小结

在本章中，我们尝试通过给出几个有趣的案例，来突出Scrapy架构的优秀性能。具体细节可能会在未来版本的Scrapy中有所变更，不过本章提供的知识应当会在很长一段时间内保持有效，并且可能会帮助你处理基于Twisted、Netty Node.js或类似框架的任何高并发异步系统。

当谈到Scrapy的性能问题时，有3个有效的答案：我不知道也不介意；我不知道但我会找出来；我知道。正如我们在本章中多次论证的，天真地回答“我们需要更多的服务器/内存/带宽”更有可能与Scrapy的性能无关。人们需要真正理解瓶颈在什么地方，并且去提升它。

在最后一章中，我们将进一步专注提升性能，通过在多台服务器上分布式部署爬虫，达到超越单机的能力。

第11章　使用Scrapyd与实时分析进行分布式爬取

我们已经走了很长的一段路。我们首先熟悉了两种基础的网络技术——HTML和XPath，然后开始使用Scrapy爬取复杂网站。接下来，我们深入了解了Scrapy通过其设置为我们提供的诸多功能，然后在探讨其Twisted引擎的内部架构和异步功能时，更加深入地了解了Scrapy和Python。在上一章中，我们研究了Scrapy的性能，并学习了如何解决复杂和经常违背直觉的性能问题。

在最后的这一章中，我将为你指出如何进一步将该技术扩展到多台服务器的一些方向。我们很快就会发现爬取工作经常是一种“高度并发”的问题，因此可以轻松地实现横向扩展，利用多台服务器的资源。为了实现该目标，我们可以像平时那样使用一个Scrapy中间件，不过也可以使用Scrapyd，这是一个专门用于管理运行在远程服务器上的Scrapy爬虫的应用。这将允许我们在自己的服务器上，拥有与第6章中介绍的相兼容的功能。

最后，我们将使用基于Apache Spark的简单系统，对抽取的数据执行实时分析。Apache Spark是一个非常流行的大数据处理框架。我们将使用Spark Streaming API展示在数据收集增多时越来越准确的结果。对于我来说，最终的这个应用展示了Python作为一种语言的能力和成熟度，因为我们只需这些，就能编写出富有表现力、简洁并且高效的代码，实现从数据抽取到分析的全栈工作。

11.1 房产的标题是如何影响价格的

我们尝试解决的示例问题是找出标题是如何与房产价格相关的。我们会认为诸如“Jacuzzi”或“pool”这样的词汇与高价位相关，而类似“discount”这样的词汇与低价位相关。结合位置信息，就可能根据该位置信息和描述，为我们提供房产是否特价的实时报警。

我们所需要计算的是给定词汇在是否存在时的价格差：

$$Shift_{term} = (\overline{Price_{properties-with-term}} - \overline{Price_{properties-without-term}})/\overline{Price}$$

比如，假设平均租金为$1000，我们观察到包含词汇jacuzzi的房产平均价格是$1300，而不包含该词汇的房产平均价格是$995，那么jacuzzi的价格差为$shift_{jacuzzi}$ = (1300−995) / 1000 = 30.5%。如果存在一个包含jacuzzi关键词的房产，其价格只比平均价格高出5%，那么我会非常想要了解它。

请注意，该指标并非微不足道，因为关键词的效果将会被聚合。例如，既包含jacuzzi又包含discount的标题很可能显示出这些关键词的组合效果。我们收集并分析的数据越多，预估的准确度越高。下面我们将回到该问题上来，讲解如何在一分钟内实现一个流媒体解决方案。

11.2 Scrapyd

现在，我们将要开始介绍Scrapyd。Scrapyd这个应用允许我们在服务器上部署爬虫，并使用它们制定爬取的计划任务。让我们来感受一下使用它是多么简单吧。我们在开发机中已经预安装了该应用，所以可以

立即使用第3章中的代码对其进行测试。我们在之前使用了几乎完全相同的过程，在这里只有一个小的变化。

首先，我们访问`http://localhost:6800/`，来看一下Scrapyd的Web界面，如图11.1所示。

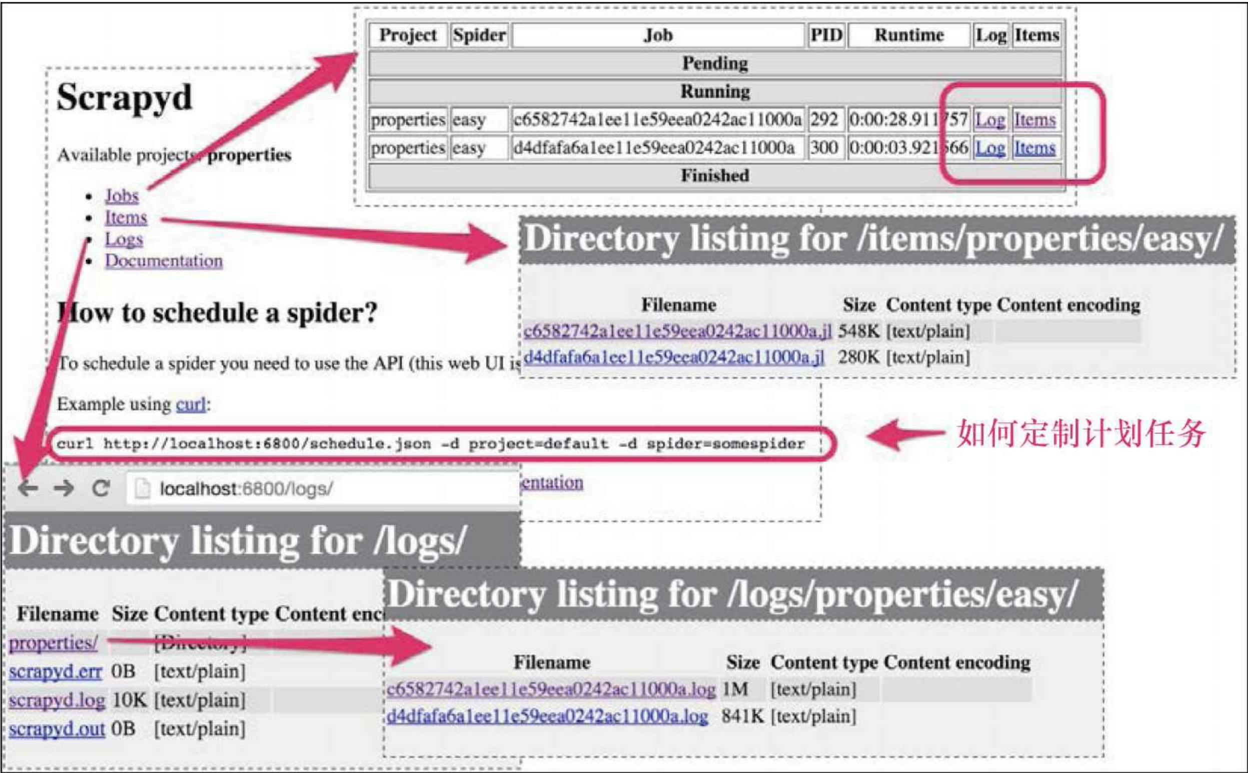


图11.1　Scrapyd的Web界面

可以看出，Scrapyd对于**Jobs**、**Items**、**Logs**和**Documentation**都有不同的区域。此外，它还提供了一些指引，告知我们如何使用其API定制计划任务。

为了完成该测试，我们必须先在Scrapyd服务器上部署爬虫。第一步是按照如下操作修改`scrapy.cfg`配置文件。

```
$ pwd

/root/book/ch03/properties

$ cat scrapy.cfg

...

[settings]

default = properties.settings

[deploy]

url = http://localhost:6800/
```

```
project = properties
```

基本上，我们所有需要做的就是去除**url**一行的注释。默认的设置已经很合适了。现在，要想部署爬虫，需要使用`scrapyd-client`提供的`scrapyd-deploy`工具。`scrapyd-client`曾经是Scrapy的一部分，不过现在已经独立为一个单独的模块，该模块可以使用`pip install scrapyd-client`安装（已经在开发机中安装好了该模块）。

```
$ scrapyd-deploy

Packing version 1450044699

Deploying to project "properties" in http://localhost:6800/addversion.

json
```

```
Server response (200):

{"status": "ok", "project": "properties", "version": "1450044699",

"spiders": 3, "node_name": "dev"}
```

当部署成功后，可以在Scrapyd的Web界面主页的**Available projects**区域看到该项目。现在，可以按照提示在该页面提交一个任务。

```
$ curl http://localhost:6800/schedule.json -d project=properties -d

spider=easy

{"status": "ok", "jobid": " d4df...", "node_name": "dev"}
```

如果回到Web界面的**Jobs** 区域，可以看到任务正在运行。稍后可以使用`schedule.json` 返回的`jobid`，通过`cancel.json` 取消该任务。

```
$ curl http://localhost:6800/cancel.json -d project=properties -d

job=d4df...

{"status": "ok", "prevstate": "running", "node_name": "dev"}
```

请一定记住执行取消操作，否则你会浪费一段时间的计算机资源。

非常好！当访问**Logs** 区域时，可以看到日志；而当访问**Items** 区域时，可以看到刚才爬取的`Item` 。这些都会在一定周期之后清空以释放

空间，因此在几次爬取操作后这些内容可能就不再可用。

如果有合理的理由，比如冲突，那么我们可以使用`http_port`修改端口，这是Scrapyd提供的诸多设置之一。通过访问`http://scrapyd.readthedocs.org/`来了解Scrapyd的文档是非常值得的。在本章中，我们需要修改的一个重要设置是`max_proc`。如果将该设置保留为默认值0的话，Scrapyd将在Scrapy任务运行时允许4倍于CPU数量的并发。由于我们将运行多个Scrapyd服务器，并且大部分可能是在虚拟机当中的，因此我们将会设置该值为4，即允许至多4个任务并发运行。这与本章的需求有关，而在实际部署当中，一般情况下使用默认值就能够良好运行。

11.3　分布式系统概述

对我来说，设计该系统是一个非常棒的经历（见图11.2）。起初，我增加了功能和复杂性，以至于不得不要求读者拥有高端硬件才能运行这些示例。这就造成之后的一个紧迫需求成为简化——无论是为了保持硬件需求更加实际，还是确保本章能够保持专注在Scrapy上。

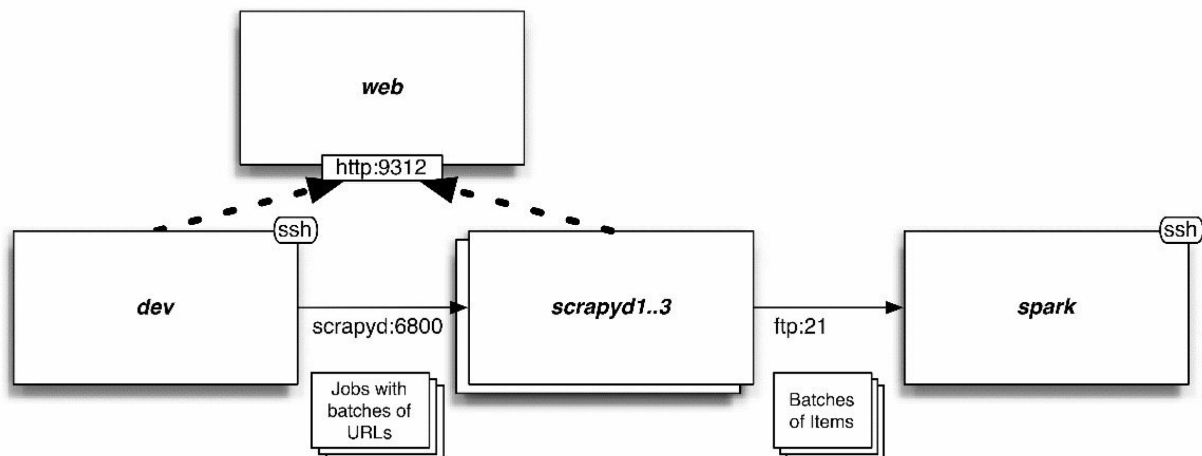


图11.2　系统概述

最后，本章将要使用的系统包含我们的开发机以及几个其他服务器。我们将使用开发机执行索引页面的垂直抓取，并从中批量抽取URL。之后，将以轮询的方式将这些URL分发到Scrapyd节点当中执行爬取。最后，包含`Item`的`.jl`文件将会通过FTP传输到运行Apache Spark的服务器中。什么？FTP？是的，我选择FTP和本地文件系统，而不是HDFS或Apache Kafka的原因是因为其内存需求很低，并且Scrapy后端的FEED_URI能够直接支持。请注意，通过简单修改Scrapyd和Spark的配置，我们可以使用Amazon S3来存储这些文件，享受其带来的冗余性、扩展性等诸多特性。不过，这里不会有更多有意思的相关话题来学习任何奇技淫巧。



使用FTP的一个风险是Spark可能会在其上传过程中看到不完整的文件。为了避免发生该问题，我们将使用Pure-FTPd以及一个回调脚本，在上传完成后立即将上传的文件移动到`/root/items`目录中。

每隔几秒，Spark将会检测该目录（`/root/items`），读取任何新文件，形成小批次，并执行分析。我们使用Apache Spark是因为它支持Python作为其编程语言，并且还支持流。到目前为止，我们可能已经使用了一些生命周期相对较短的爬取工作，不过现实世界中许多爬取工作永远都不会结束。爬取工作24/7不间断运行，并提供用于分析的数据流，数据越多其结果就越精确。正因如此，我们将使用Apache Spark进行展示。



使用Apache Spark和Scrapy并没有什么特殊之处。你也可以选择使用Map-Reduce、Apache Storm或任何其他适合你需求的框架。

在本章中，我们并不会将`Item`插入到诸如ElasticSearch或MySQL等数据库当中。第9章中介绍的技术在这里同样适用，不过其性能会很糟糕。当你每秒钟执行数千次写入操作时，只有极少数的数据库系统能够运行良好，但这正是我们的管道将会做的事情。如果我们想要向数据库中插入数据，则需要遵循与使用Spark相似的流程，即批量导入生成的`Item`文件。你可以修改我们的Spark示例流程，批量导入到任意数据库当中。

最后需要注意的是，该系统并没有良好的弹性。我们假设各节点都是健康的，并且任何失败都不会产生严重的业务影响。Spark拥有弹性配置，能够提供高可用性。而除了Scrapyd的持久化队列外，Scrapy并没有提供任何相关的内建功能，这就意味着失败的任务需要在节点恢复后

才能重新启动。这种方式对于你的需求来说，也许适合，也许不适合。如果对你而言弹性十分重要，那么你需要搭建监控和分布式队列方案（如基于Kafka或RabbitMQ），来重启失败的爬取工作。

11.4　爬虫和中间件的变化

为了构建该系统，我们需要稍微对Scrapy爬虫进行修改，并且需要开发爬虫中间件。更具体地说，我们必须执行如下操作：

- 调整索引页爬取，以最大速率执行；
- 编写中间件，分批发送URL到Scrapyd服务器；
- 使用相同中间件，允许在启动时使用批量URL。

我们将尝试使用尽可能小的改动来实现这些变化。理想情况下，整个操作应该清晰、易理解并且对其依赖的爬虫代码透明。这应该是一个基础架构层级的需求，如果想对爬虫（可能数百个）进行修改来实现它则是一个坏主意。

11.4.1　索引页分片爬取

我们的第一步是优化索引页爬取，使其尽可能更快。在开始之前，先来设置一些期望。假设爬虫爬取并发量是16，并且我们测量得到其与源网站服务器的延迟大约为0.25秒。此时得到的吞吐量最多为16 / 0.25 = 64页/秒。索引页数量为50000个详情页 / 每个索引页30个详情页链接 = 1667索引页。因此，我们期望索引页下载花费的时间大约为1667 / 64 = 26秒多一点。

让我们以第3章中名为easy 的爬虫开始。先把执行垂直抓取的Rule注释掉（callback='parse_item' 的那个），因为现在只需要爬取索引页。



你可以在GitHub中获取到本书的全部代码。下载该代码，可以访问：git clone https://github.com/scalingexcellence/scrapybook 。

本章中的完整代码位于ch11 目录当中。

如果我们在进行任何优化之前对scrapy crawl 只爬取10个页面的情况进行计时，可以得到如下结果。

```
$ ls

properties scrapy.cfg

$ pwd

/root/book/ch11/properties
```

```
$ time scrapy crawl easy -s CLOSESPIDER_PAGECOUNT=10

...

DEBUG: Crawled (200) <GET ...index_00000.html> (referer: None)

DEBUG: Crawled (200) <GET ...index_00001.html> (referer: ...index_00000.

html)

...

real 0m4.099s
```

如果10个页面就花费了4秒时间，那么就不可能在26秒时间内完成1,700个页面。通过查看日志，我们发现每个页面都来自于前一个页面的下一页链接，也就是说在任意时刻都只有至多一个页面正在执行爬取。我们的有效并发为1。我们希望并行处理，得到想要的并发数量（16个并发请求）。我们将对索引分片，并允许一些额外的分片，以确保爬虫中的URL不会不足。我们将会把索引分为20个段。实际上，任何超过16的数值都能够增加速度，不过在超过20之后所得到的回报呈递减趋势。我们将通过如下表达式计算每个分片的起始索引ID。

```
>>> map(lambda x: 1667 * x / 20, range(20))

[0, 83, 166, 250, 333, 416, 500, ... 1166, 1250, 1333, 1416, 1500, 1583]
```

因此，我们使用如下代码设置`start_urls` 。

```
start_urls = ['http://web:9312/properties/index_%05d.html' % id
```

```
for id in map(lambda x: 1667 * x / 20, range(20))]
```

这可能和你的索引有很大的不同，因此我们没必要在此处实现得更漂亮。如果还设定了并发设置（CONCURRENT_REQUESTS、CONCURRENT_REQUESTS_PER_DOMAIN ）为16，那么当运行爬虫时，将会得到如下结果。

```
$ time scrapy crawl easy -s CONCURRENT_REQUESTS=16 -s CONCURRENT_

REQUESTS_PER_DOMAIN=16

...

real 0m32.344s
```

该结果已经与期望值非常接近了。我们的下载速度为 1667个页面 / 32秒 = 52个索引页/秒，这就意味着每秒可以生成52×30 = 1560个详情页URL。现在，可以将垂直抓取的`Rule` 的注释取消掉，保存文件作为新爬虫分发。我们不需要对爬虫代码进行更多修改，这显示出我们即将开发的中间件的强大以及非侵入性。如果只使用开发服务器运行`scrapy crawl`，假设处理详情页的速度和索引页处理时一样快，那么它将花费不少于50000 / 52 = 16分钟时间完成爬取。

本节有两个关键内容。在学习完第10章之后，我们已经可以实现真正的工程。我们能够精确计算出系统期望得到的性能，并且确保在达到该性能之前不会停止（在合理范围内）。第二个要记住的重要事情是，由于索引页爬取提供了详情页，爬取的总吞吐量将会是其吞吐量的最小值。如果我们生成的URL比Scrapyd能够消费得更快，那么URL将会堆积在其队列当中。反过来，如果生成的URL太慢，Scrapyd将会拥有过剩的无法利用的能力。

11.4.2　分批爬取URL

现在，我们准备开发处理详情页URL的基础架构，目的是对其进行垂直爬取、分批并分发到多台Scrapyd节点中，而不是在本地爬取。

如果查看第8章中的Scrapy架构，就可以很容易地得出结论，这是

爬虫中间件的任务，因为它实现了process_spider_output()，在到达下载器之前，在此处处理请求，并能够中止它们。我们在实现中限制只支持基于CrawlSpider的爬虫，另外还只支持简单的GET请求。如果需要更加复杂，比如POST或有权限验证的请求，那么需要开发更复杂的功能来扩展参数、请求头，甚至可能在每次批量运行后重新登录。

在开始之前，先来快速浏览一下Scrapy的GitHub。我们将回顾SPIDER_MIDDLEWARES_BASE设置，以查看Scrapy提供的参考实现，以便尽最大可能复用它。Scrapy 1.0包含如下爬虫中间件：HttpErrorMiddleware、OffsiteMiddleware、RefererMiddleware、UrlLengthMiddleware以及DepthMiddleware。在快速了解它们的实现之后，我们发现OffsiteMiddleware（只有60行代码）与想要实现的功能很相似。它根据爬虫的allowed_domains属性，把URL限制在某些特定域名中。我们可以使用相似的模式吗？和OffsiteMiddleware实现中丢弃URL不同，我们将对这些URL进行分批并发送到Scrapyd节点中。事实证明这是可以的。下面是实现的部分代码。

```
def __init__(self, crawler):
    settings = crawler.settings
    self._target = settings.getint('DISTRIBUTED_TARGET_RULE', -1)
    self._seen = set()
    self._urls = []
    self._batch_size = settings.getint('DISTRIBUTED_BATCH_SIZE', 1000)
    ...

def process_spider_output(self, response, result, spider):
    for x in result:
        if not isinstance(x, Request):
            yield x
        else:
            rule = x.meta.get('rule')
```

```
            if rule == self._target:
                self._add_to_batch(spider, x)
            else:
                yield x

def _add_to_batch(self, spider, request):
    url = request.url
    if not url in self._seen:
        self._seen.add(url)
        self._urls.append(url)
        if len(self._urls) >= self._batch_size:
            self._flush_urls(spider)
```

process_spider_output()既处理Item也处理Request。我们只想处理Request，因此我们对其他所有内容执行yield操作。如果查看CrawlSpider的源代码，就会注意到将Request / Response映射到Rule的方式是通过其meta字典的名为'rule'的整型字段。我们检查该数值，如果它指向目标的Rule（DISTRIBUTED_TARGET_RULE设置），则会调用_add_to_batch()添加URL到当前批次。然后，丢弃该Request。对其他所有Request执行yield操作，比如下一页链接、无变化的链接。_add_to_batch()方法实现了一个去重机制。不过很遗憾的是，由于前一节中描述的分片流程，我们可能对少数URL抽取两次。我们使用_seen集合检测并丢弃重复值。然后，把这些URL添加到_urls列表中，如果其大小超过_batch_size（DISTRIBUTED_BATCH_SIZE设置），就会触发调用_flush_urls()。该方法提供了如下的关键功能。

```
def __init__(self, crawler):
    ...
```

```
    self._targets = settings.get("DISTRIBUTED_TARGET_HOSTS")
    self._batch = 1
    self._project = settings.get('BOT_NAME')
    self._feed_uri = settings.get('DISTRIBUTED_TARGET_FEED_URL', None)
    self._scrapyd_submits_to_wait = []

def _flush_urls(self, spider):
    if not self._urls:
        return

    target = self._targets[(self._batch-1) % len(self._targets)]

    data = [
         ("project", self._project),
         ("spider", spider.name),
         ("setting", "FEED_URI=%s" % self._feed_uri),
         ("batch", str(self._batch)),
    ]

    json_urls = json.dumps(self._urls)
    data.append(("setting", "DISTRIBUTED_START_URLS=%s" % json_urls))

    d = treq.post("http://%s/schedule.json" % target,
                  data=data, timeout=5, persistent=False)

    self._scrapyd_submits_to_wait.append(d)

    self._urls = []
    self._batch += 1
```

首先，它使用一个批次计数器（`_batch`）来决定要将该批次发送到哪个Scrapyd服务器中。我们在`_targets`（`DISTRIBUTED_TARGET_HOSTS`设置）中保持更新可用的服务器。然后，构造POST请求到Scrapyd的`schedule.json`。这比之前通过`curl`执行的更加高级，因为它传递了一些精心挑选的参数。基于这些参数，Scrapyd可以有效地计划运行任务，类似如下所示。

```
scrapy crawl distr \
-s DISTRIBUTED_START_URLS='[".../property_000000.html", ... ]' \
-s FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl' \
-a batch=1
```

除了项目和爬虫名外，我们还向爬虫传递了一个FEED_URI设置。我们可以从DISTRIBUTED_TARGET_FEED_URL设置中获取该值。

由于Scrapy支持FTP，我们可以让Scrapyd通过匿名FTP的方式将爬取到的`Item` 文件上传到Spark服务器中。格式包含爬虫名（`%(name)s` ）和时间（`%(time)s` ）。如果只使用这些，那么当两个文件的创建时间相同时，最终会产生冲突。为了避免意外覆盖，我们还添加了一个`%(batch)s` 参数。默认情况下，Scrapy不知道任何关于批次的事情，因此我们需要找到一种方式来设置该值。Scrapyd中`schedule.json` 这个API的一个有趣特性是，如果参数不是设置或少数几个已知参数的话，它将会被作为参数传给爬虫。默认情况下，爬虫参数将会成为爬虫属性，未知的`FEED_URI` 参数将会去查阅爬虫的属性。因此，通过传递`batch` 参数给`schedule.json` ，我们可以在`FEED_URI` 中使用它以避免冲突。

最后一步是使用编码为JSON的该批次详情页URL编译为`DISTRIBUTED_ START_URLS` 设置。除了熟悉和简单之外，使用该格式并没有什么特殊的理由。任何文本格式都可以做到。



> 通过命令行向Scrapy传输大量数据丝毫也不优雅。在一些时候，你想要将参数存储到数据存储中（比如Redis），并且只向Scrapy传输ID。如果想要这样做，则需要在`_flush_urls()`和`process_start_requests()`中做一些小的改变。

我们使用`treq.post()`处理POST请求。Scrapyd对持久化连接处理得不是很好，因此使用`persistent=False`禁用该功能。为了安全起见，我们还设置了一个5秒的超时时间。有趣的是，我们为该请求在`_scrapyd_submits_to_wait`列表中存储了延迟函数，后续内容中将会进行讲解。关闭该函数时，我们将重置`_urls`列表，并增加当前的`_batch`值。

出人意料的是，我们在关闭操作处理器中发现了如下所示的诸多功能。

```
def __init__(self, crawler):
    ...
    crawler.signals.connect(self._closed, signal=signals.spider_
closed)

@defer.inlineCallbacks
def _closed(self, spider, reason, signal, sender):
    # Submit any remaining URLs
    self._flush_urls(spider)

    yield defer.DeferredList(self._scrapyd_submits_to_wait)
```

`_close()`将会在我们按下*Ctrl* + *C*或爬取完成时被调用。无论哪种情况，我们都不希望丢失属于最后一个批次的任何URL，因为它们还

没有被发送出去。这就是为什么我们在_close()方法中首先要做的是调用_flush_urls(spider)清空最后的批次的原因。第二个问题是，作为非阻塞代码，任何treq.post()在停止爬取时都可能完成或没有完成。为了避免丢失任何批次，我们将使用之前提及的scrapyd_submits_to_wait列表，来包含所有的treq.post()的延迟函数。我们使用defer.DeferredList()进行等待，直到全部完成。由于_close()使用了@defer.inlineCallbacks，我们只需对其执行yield操作，并在所有请求完成之后进行恢复即可。

总结来说，在DISTRIBUTED_START_URLS设置中包含批量URL的任务将被送往Scrapyd服务器，并在这些Scrapyd服务器中运行相同的爬虫。很明显，我们需要某种方式以使用该设置初始化start_urls。

11.4.3　从设置中获取初始URL

当你注意到爬虫中间件提供的用于处理爬虫给我们的start_requests的process_start_requests()方法时，就会感受到爬虫中间件是怎样满足我们的需求的。我们检测DISTRIBUTED_START_URLS设置是否已被设定，如果是的话，则解码JSON并使用其中的URL对相关的Request进行yield操作。对于这些请求，我们设置CrawlSpider的_response_download()方法作为回调，并设置meta['rule']参数，以便其Response能够被适当的Rule处理。坦白来说，我们查阅了Scrapy的源代码，发现CrawlSpider创建Request的方式使用了相同的方法。在本例中，代码如下所示。

```
def __init__(self, crawler):
```

```
    ...
    self._start_urls = settings.get('DISTRIBUTED_START_URLS', None)
    self.is_worker = self._start_urls is not None

def process_start_requests(self, start_requests, spider):
    if not self.is_worker:
        for x in start_requests:
            yield x
    else:
        for url in json.loads(self._start_urls):
            yield Request(url, spider._response_downloaded,
                          meta={'rule': self._target})
```

我们的中间件已经准备好了。可以在settings.py 中启用它并进行设置。

```
SPIDER_MIDDLEWARES = {
    'properties.middlewares.Distributed': 100,
}
DISTRIBUTED_TARGET_RULE = 1
DISTRIBUTED_BATCH_SIZE = 2000
DISTRIBUTED_TARGET_FEED_URL = ("ftp://anonymous@spark/"
                               "%(batch)s_%(name)s_%(time)s.jl")
DISTRIBUTED_TARGET_HOSTS = [
    "scrapyd1:6800",
    "scrapyd2:6800",
    "scrapyd3:6800",
]
```

一些人可能会认为DISTRIBUTED_TARGET_RULE 不应该作为设置，因为不同爬虫之间可能是不一样的。你可以将其认为是默认值，并且可以在爬虫中使用custom_settings 属性进行覆写，比如：

```
custom_settings = {
    'DISTRIBUTED_TARGET_RULE': 3
}
```

不过在我们的例子中并不需要这么做。我们可以做一个测试运行，爬取作为设置提供的单个页面。

```
$ scrapy crawl distr -s \

DISTRIBUTED_START_URLS='["http://web:9312/properties/property_000000.html"
]'
```

当爬取成功后，可以尝试更进一步，爬取页面后使用FTP传输给Spark服务器。

```
scrapy crawl distr -s \

DISTRIBUTED_START_URLS='["http://web:9312/properties/property_000000.
```

```
html"]' \

-s FEED_URI='ftp://anonymous@spark/%(batch)s_%(name)s_%(time)s.jl' -a batc
h=12
```

如果你通过ssh登录到Spark服务器中（稍后会有更多介绍），将会看到一个文件位于`/root/items`目录中，比如`12_distr_date_time.jl`。

上述是使用Scrapyd实现分布式爬取的中间件的示例实现。你可以使用它作为起点，实现满足自己特殊需求的版本。你可能需要适配的事情包括如下内容。

- 支持的爬虫类型。比如，一个不局限于`CrawlSpider`的替代方案可能需要你的爬虫通过适当的`meta`以及采用回调命名约定的方式来标记分布式请求。
- 向Scrapyd传输URL的方式。你可能希望使用特定域名信息来减少传输的信息量。比如，在本例中，我们只传输了房产的ID。
- 你可以使用更优雅的分布式队列解决方案，使爬虫能够从失败中恢复，并允许Scrapyd将更多的URL提交到批处理。

- 你可以动态填充目标服务器列表，以支持按需扩展。

11.4.4　在Scrapyd服务器中部署项目

为了能够在我们的3台Scrapyd服务器中部署爬虫，我们需要将这3台服务器添加到`scrapy.cfg`文件中。该文件中的每个`[deploy:target-name]`区域都定义了一个新的部署目标。

```
$ pwd

/root/book/ch11/properties

$ cat scrapy.cfg

...

[deploy:scrapyd1]

url = http://scrapyd1:6800/

```

```
[deploy:scrapyd2]

url = http://scrapyd2:6800/

[deploy:scrapyd3]

url = http://scrapyd3:6800/
```

可以通过scrapyd-deploy -l 查询当前可用的目标。

```
$ scrapyd-deploy -l

scrapyd1            http://scrapyd1:6800/
```

```
scrapyd2            http://scrapyd2:6800/

scrapyd3            http://scrapyd3:6800/
```

通过scrapyd-deploy <target-name>，可以很容易地部署任意服务器。

```
$ scrapyd-deploy scrapyd1

Packing version 1449991257

Deploying to project "properties" in http://scrapyd1:6800/addversion.json

Server response (200):
```

```
{"status": "ok", "project": "properties", "version": "1449991257",

"spiders": 2, "node_name": "scrapyd1"}
```

该过程会留给我们一些额外的目录和文件（build、project.egg-info、setup.py），我们可以安全地删除它们。本质上，scrapyd-deplo``y 所做的事情就是打包你的项目，并使用addversion.jso n上传到目标Scrapyd服务器当中。

之后，当我们使用scrapyd-deploy -L 查询单台服务器时，可以确认项目是否已经被成功部署，如下所示。

```
$ scrapyd-deploy -L scrapyd1

properties
```

我还在项目目录中使用`touch`创建了3个空文件（`scrapyd1-3`）。使用`scrapyd*`扩展为文件名称，同样也是目标服务器的名称。之后，你可以使用一个bash循环部署所有服务器：`for i in scrapyd*; do scrapyd-deploy $i; done`。

11.5　创建自定义监控命令

如果想监控多台Scrapyd服务器的爬虫进程，则需要手动执行。这是一个很好的机会，能够让我们练习到目前为止所见到的一切知识，创建一个原始的Scrapy命令——`scrappy monitor`，用于监控一组Scrapyd服务器。我们将该文件命名为`monitor.py`，并且在`settings.py`文件中添加`COMMANDS_MODULE = 'properties.monitor'`。通过快速浏览Scrapyd的文档，我们发现`listjobs.json`这个API可以为我们提供任务相关的信息。如果想要找到给定目标的基础URL，可以猜到它一定在`scrapyd-deploy`代码中的某个地方，从而可以让我们在单个文件中找到它。如果查看`https://github.com/scrapy/scrapyd-client/blob/master/scrapyd-client/scrapyd-deploy`，很快就会发现`_get_target()`函数（由于其实现没有添加太多值，因此我会忽略它），在该函数中将会给我们提供目标名称及其基础URL。太棒了！我们开始实现该命令的第一部分吧，其代码如下所示。

```
class Command(ScrapyCommand):
    requires_project = True

    def run(self, args, opts):
        self._to_monitor = {}
        for name, target in self._get_targets().iteritems():
            if name in args:
                project = self.settings.get('BOT_NAME')
                url = target['url'] + "listjobs.json?project=" + project
                self._to_monitor[name] = url

        l = task.LoopingCall(self._monitor)
        l.start(5) # call every 5 seconds

        reactor.run()
```

目前我们所看到的实现还是很简单的。它使用目标名称和我们想要监控的API地址填充`_to_monitor`字典。然后，我们使用`task.LoopingCall()`计划到`_monitor()`方法的定期调用。`_monitor()`方法使用了`treq`和延迟操作，而我们使用了`@defer.inlineCallbacks`来简化其实现。下面是其代码（已忽略一些错误处理和装饰）。

```
@defer.inlineCallbacks
def _monitor(self):
    all_deferreds = []
    for name, url in self._to_monitor.iteritems():
        d = treq.get(url, timeout=5, persistent=False)
        d.addBoth(lambda resp, name: (name, resp), name)
        all_deferreds.append(d)

    all_resp = yield defer.DeferredList(all_deferreds)

    for (success, (name, resp)) in all_resp:
        json_resp = yield resp.json()
        print "%-20s running: %d, finished: %d, pending: %d" %
              (name, len(json_resp['running']),
```

```
        len(json_resp['finished']), len(json_resp['pending']))
```

上面这些行已经包含了我们知道的几乎所有Twisted技术。我们使用`treq` 调用Scrapyd的API，并且使用`defer.DeferredList` 立即处理所有响应。当我们的所有结果进入到`all_resp` 之后，则开始迭代并获取其JSON对象。`treq Response` 的`json()` 方法将会返回延迟操作，而不是真实值，我们对其执行了`yield` 操作，并会在未来的某个时间点恢复其真实值。最后一步，我们打印出结果。JSON响应包含待处理、运行中及已完成任务列表的信息，我们将打印出它们的长度。

11.6　使用Apache Spark流计算偏移量

此刻，我们的Scrapy系统功能齐全。现在，让我们快速看一下Apache Spark的功能。

在本章最开始介绍的公式$shift_{term}$ 非常简单好用，但是无法有效实现。我们可以通过两个计数器计算$\overline{Price}$，使用$2 \cdot n_{terms}$ 个计数器计算$\overline{Price_{with}}$，每个新价格只需更新其中的4个。不过计算$\overline{Price_{without}}$ 则是一个很大的问题，因为对于每个新价格来说，都需要更新$2 \cdot (n_{terms} - 1)$个计数器。比如，我们需要添加jacuzzi的价格到每个$Price_{without}$ 计数器中，而不是只有jacuzzi这一个。这会造成算法由于包含大量条件而不可行。

为了解决该问题，我们所能注意到的是，如果我们将带某个条件的房产价格，与不带相同条件的房产价格相加，将会得到所有房产的价格

（很明显！），即$\Sigma Price = \Sigma Price|_{with} + \Sigma Price|_{without}$。因此，不带某个条件的房产平均价格可以使用如下的代价很小的操作进行计算。

$$\overline{Price_{without}} = \frac{\sum Price_{without}}{n_{without}} = \frac{\sum Price - \sum Price|_{without}}{n - n_{with}}$$

使用该公式，偏移公式变为如下所示。

$$Shift_{term} = \left(\frac{\sum Price|_{with}}{n_{with}} - \frac{\sum Price - \sum Price|_{with}}{n - n_{with}} \right) / \frac{\sum Price}{n}$$

现在让我们看看如何实现该公式。请注意此处不是Scrapy的代码，因此感到有些陌生是很正常的，不过你仍然可以不费太多力气就能阅读并理解该代码。你可以在`boostwords.py`中找到该应用。请记住该代码中包含很多复杂的测试代码，你可以安全地忽略它们。其核心代码如下所示。

```
# Monitor the files and give us a DStream of term-price pairs
raw_data = raw_data = ssc.textFileStream(args[1])
word_prices = preprocess(raw_data)

# Update the counters using Spark's updateStateByKey
running_word_prices = word_prices.updateStateByKey(update_state_function)

# Calculate shifts out of the counters
shifts = running_word_prices.transform(to_shifts)

# Print the results
shifts.foreachRDD(print_shifts)
```

Spark使用所谓的`DStream`表示数据流。`textFileStream()`方法监控文件系统的目录，当它检测到新文件时，将会从中获取数据

流。preprocess()函数将其转变为条件/价格对的数据流。我们通过Spark的updateStateByKey()方法，使用update_state_function()函数，在运行的计数器中聚合这些条件/价格对。最后，通过运行to_shifts()计算偏移量，并使用print_shifts()函数打印出最佳结果。我们的大部分功能都很简单，它们只是按照对Spark高效的方式形成数据。最有意思的例外是我们的to_shifts()函数。

```
def to_shifts(word_prices):
    (sum0, cnt0) = word_prices.values().reduce(add_tuples)
    avg0 = sum0 / cnt0

    def calculate_shift((isum, icnt)):
        avg_with = isum / icnt
        avg_without = (sum0 - isum) / (cnt0 - icnt)
        return (avg_with - avg_without) / avg0

    return word_prices.mapValues(calculate_shift)
```

它如此紧密地遵循公式，令人印象非常深刻。除了其简单性之外，Spark的mapValues()使我们的（可能多台）Spark服务器能够以最小网络开销高效运行calculate_shift。

11.7　运行分布式爬取

我通常使用4个终端查看爬取的完成进度。为了使本节自成一体，因此我还为你提供了打开到相关服务器终端的vagrant ssh命令（见图11.3）。

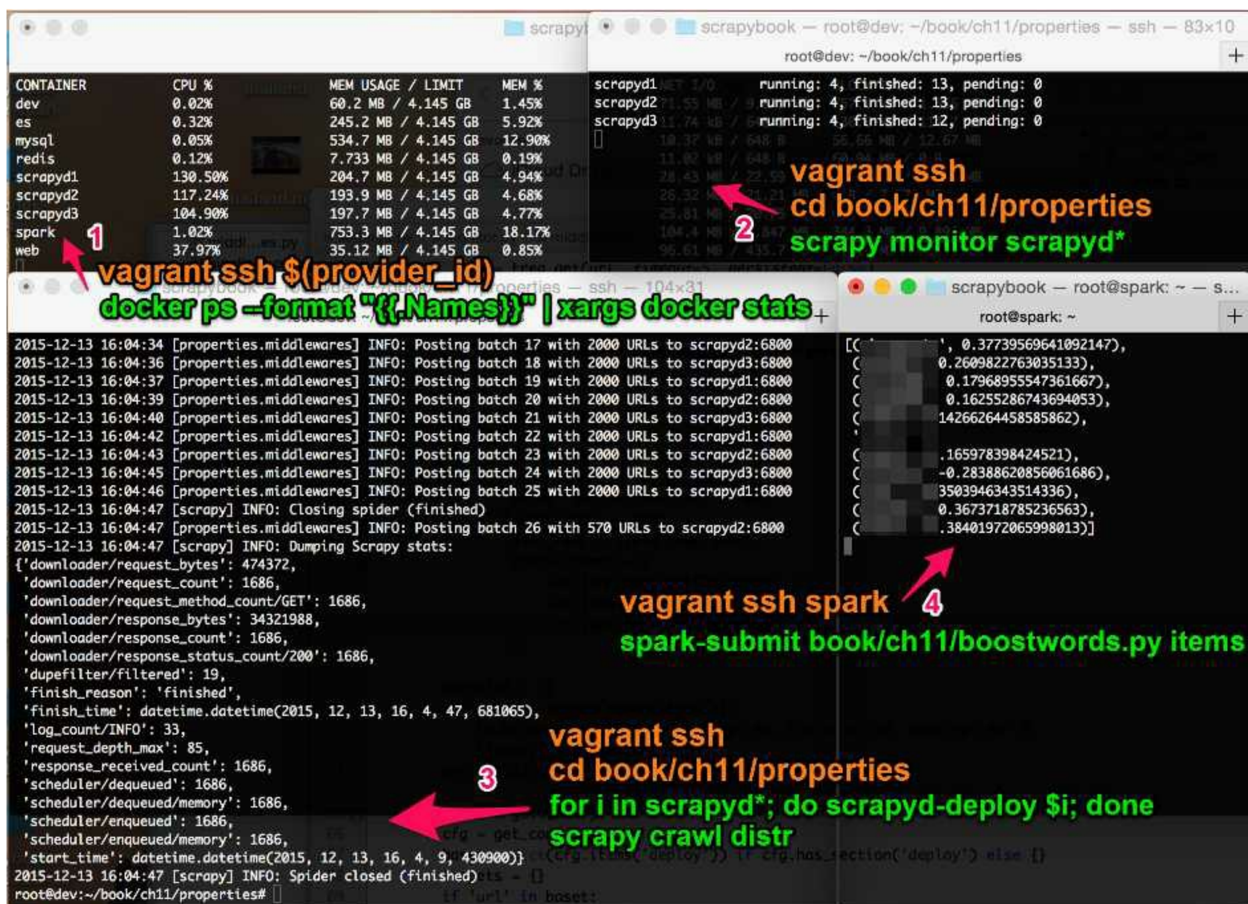


图11.3　使用4个终端监控爬取

在终端1中，我喜欢监控多台服务器的CPU和内存使用率。这有助于识别和修复潜在问题。要想启动它，可运行如下命令。

```
$ alias provider_id="vagrant global-status --prune | grep 'docker-

provider' | awk '{print \$1}'"

$ vagrant ssh $(provider_id)
```

```
$ docker ps --format "{{.Names}}" | xargs docker stats
```

前面两行稍微复杂的代码允许通过ssh登录到docker provider VM中。如果使用的不是虚拟机，而是运行在docker驱动的Linux机器上，那么只需要最后一行。

第2个终端同样用于诊断，一般按照如下命令使用它运行`scrapy monitor`。

```
$ vagrant ssh

$ cd book/ch11/properties

$ scrapy monitor scrapyd*
```

```
```

请记住使用scrapyd* 以及以服务器名称命名的空文件，scrapy monitor scrapyd* 将被扩展为scrapy monitor scrapyd1 scrapyd2 scrapyd3 。

第3个终端是我们的开发机，我们在这里启动爬虫。除此之外，大部分时间是空闲的。如果想要启动一个新的爬虫，可以执行如下命令。

```
$ vagrant ssh

$ cd book/ch11/properties

$ for i in scrapyd*; do scrapyd-deploy $i; done

$ scrapy crawl distr
```

最后两行是最基本的。首先，我们使用`for`循环及`scrapyd-deploy`部署爬虫到服务器中。然后，使用`scrapy crawl distr`启动爬取操作。我们也可以运行更少的爬取操作，比如`scrapy crawl distr -s CLOSESPIDER_PAGECOUNT=100`，以爬取大约100个索引页，相当于大概3000个详情页。

最后的第4个终端用于连接Spark服务器，我们将使用它运行数据流分析任务。

```
$ vagrant ssh spark

$ pwd

/root

$ ls

book items
```

```
$ spark-submit book/ch11/boostwords.py items
```

只有最后一行是最基本的，在该行中运行了boostwords.py，并将我们本地的items 目录提供给监控。有时，我还会使用watch ls -l items 来关注Item文件的到达情况。

究竟哪些关键词对价格影响最大呢？我把它作为惊喜，留给那些一直跟随下来的读者们。

11.8　系统性能

在性能方面，结果很大程度上取决于我们的硬件情况，以及我们给虚拟机的CPU数量和内存大小。在实际部署中，我们可以获得水平的伸缩性，可以让我们以服务器允许的最快速度运行爬取。

对于给定设置情况下的理论最大值是：3个服务器 · 4个处理器/服务器 · 16个并发请求 · 4个页面/秒（通过页面下载延迟定义）= 768个页面/秒。

实践时，在Macbook Pro中使用分配了4GB内存以及8核CPU的

VirtualBox虚拟机，我可以在2分40秒的时间内获取50,000个URL，也就是大约315个页面/秒。在拥有2个vCPU和8GB内存的Amazon EC2 m4.large实例中，由于有限的CPU能力，花费了6分12秒的时间，即134个页面/秒。在拥有16个vCPU和64GB内存的Amazon EC2 m4.4xlarge实例中，爬取完成时间是1分44秒，即480个页面/秒。在同一台机器中，我将Scrapyd的实例数量加倍，即增加到6个（只需编辑`Vagrantfile`、`scrapy.cfg`以及`settings.py`），此时爬虫完成时间为1分15秒，即其速度为667个页面/秒。在最后一种情况下，我们的Web服务器似乎遇到了瓶颈（在实际中意味着中断）。

我们得到的性能与理论最大值之间的距离是合理的。有很多小的延迟在我们的粗略计算中是没有考虑进去的。尽管我们之前声称有250ms的页面加载延迟，但是在前面的章节中可以看到该延迟实际上更大，因为至少还有Twisted和操作系统的延迟。另外，还有一些其他延迟，比如URL从开发机传输到Scrapyd服务器的时间、我们爬取的`Item`通过FTP传给Spark的时间以及Scrapyd发现和计划任务所花费的时间（平均2.5秒——参考Scrapyd的`poll_interval`设置）。此外，还有开发机以及Scrapyd爬取的启动时间没有计算进来。我将不会尝试改善这些延迟中的任何一个，除非能确定它们可以提升吞吐量。我的下一步是增加爬取的大小（比如50万个页面）、负载均衡几个Web服务器实例以及在我们的扩展尝试中发现下一个有趣的挑战。

11.9　关键要点

本章最重要的要点是，如果你想运行分布式爬虫，则应当使用合适

的批次大小。

根据源网站的响应速度，你可能有数百、数千甚至数万个URL。你会希望它们足够大，达到几分钟的级别，以便能够分摊启动成本。而另一方面，你又不希望它们过大，因为这将会使机器故障成为主要风险。在容错分布式系统中，你可以重试失败的批次，但你不会希望这将给你带来几个小时的工作量。

11.10　本章小结

我希望你能喜欢这本关于Scrapy的书，就像我编写它那样。你现在已经对Scrapy的能力有了非常丰富的了解，并且能够使用它实现或简单或复杂的爬虫场景。你也会对使用这样一个高性能系统并充分利用它进行开发的复杂性有所了解。使用爬虫，你可以通过自己的应用及时获取现实世界中的大规模数据集。我们已经看到了使用Scrapy数据集构建手机应用及实现有趣分析的方式。希望你能使用Scrapy开发出优秀、创新的应用，让我们的世界变得更好。祝你好运！

附录A　必备软件的安装与故障排除

A.1　必备软件的安装

本书使用了庞大的虚拟服务器系统演示现实中多服务器部署环境下的Scrapy使用。我们使用了行业标准工具——Vagrant和Docker，来搭建该系统。由于本书严重依赖于网站内容和布局，如果我们使用不可控的网站，那么我们的例子将会在几个月的时间之后无法使用。Vagrant和Docker为我们提供了一个独立的环境，在这里我们的示例无论现在还是以后都能正常运行。作为附带的好处，我们不会访问任何远程服务器，因此就不会对任何网站管理者造成不便。即使我们破坏了某些东西，造成示例无法工作，也可以使用两个命令：`vagrant destroy` 和 `vagrant up --no-parallel`，销毁并重建系统，继续运行。

在开始之前，我需要说明一下，该基础架构是专门为本书读者的需求定制的。尤其是有关Docker的部分，普遍共识是每个Docker容器应当是只运行单一进程的微服务。我们并没有这么做。我们的很多Docker容器都比较重，我们可以使用`vagrant ssh`连接它们并执行各种操作。尤其是我们的开发机看起来一点也不像微服务。这是我们去往该隔离系统的用户友好的网关，我们将其视为功能齐全的Linux机器。如果我们不使用这种方式改变规则，就必须使用大量的Vagrant和Docker命令，更加深入地排查故障，在这种情况下本书将很快变为Vagrant/Docker书籍。我希望Docker爱好者能够原谅我们，并且每位读者能够享受到

Vagrant和Docker带给我们的方便和益处。



本书中的容器不适用于生产环境。

我们不可能测试每个软件/硬件的配置。假设某些地方无法工作，如果可以的话，请修复它并在GitHub中向我们发送一个Pull Request。如果你不知道如何修复，那么请在GitHub上搜索相关issue，如果不存在的话请打开一个新的issue。

A.2 系统

本节用于参考。你可以先跳过本节内容，当想要更好地理解本书系统的构成方式时，可以返回来阅读本节。我们在相关章节中重复了本节中的部分信息。

我们使用Vagrant构建了如下系统（见图A.1）。

图A.1　本书使用的系统

在图A.1中，每个方框表示一台服务器，主机名是其标题的第一部分（`dev`、`Web`、`es`等）。标题的第二部分是其使用的Docker镜像（`scrapybook/dev`、`scrapybook/web`、`scrapybook/es`等）。下面是运行在该服务器上的软件的简要描述。线段表示不同服务器之间的链接，其协议写在线段旁边。Docker所提供的隔离的一部分是不允许超出显式声明的连接。也就是说，比如你想在Spark服务器上使用1234端口监听某些东西，除非你在Vagrant文件中添加相关声明暴露该端口，否则没有人能连接到该端口。请记住这一点，以避免在其他服务器中安装自定义软件时出现问题。

在大部分章节中，我们只会使用到两个机器：dev 和web 。vagrant ssh 可以让我们连接到开发机中。我们可以从这里使用主机名很轻松地访问其他机器（mysql 、web 等）。我们可以通过执行如ping web 的操作来确认能否访问web机器。我们在每章中使用并解释了很多命令。第9章演示了如何推送数据到不同的数据库。第11章使用了3个运行Scrapyd的Docker容器（实际上与开发机相同，以减少下载大小），这些机器的主机名分别是scrapyd1-3 。我们还使用了一个主机名为spark 的服务器，用于运行Apache Spark以及FTP服务。可以使用vagrant ssh spark 连接该服务器，并运行Spark任务。

可以在GitHub顶级目录的Vagrantfile 中找到该系统的描述。当输入vagrant up --no-parallel 时，系统将开始构建。这将会花费几分钟时间，尤其是在第一次构建时，我们将会在后面的FAQ中了解到更详细的介绍。可以看到，本书代码是挂载在~/book 目录当中的。任何时候我们在宿主机修改其中的内容时，变更都会自动传播。这样我们就可以使用文本编辑器或IDE修改文件，并且可以在开发机中快速查看变化了。

最后，一些监听端口被转发到我们的宿主机中，并暴露了相关的服务。比如，你可以使用一个简单的Web浏览器来访问它们。如果你已经在计算机中使用了其中某个端口，那么会产生冲突，导致系统构建无法成功。我们将会在后面的FAQ中告知你如何解决这种情况。表A.1是转发的端口列表。

表**A.1**

机器和服务	从开发机访问的地址	从你的（宿主）机访问的地址
Web—eb服务器	http://web:9312	http://localhost:9312
dev—scrapyd	http://dev:6800	http://localhost:6800
scrapyd1—scrapyd	http://scrapyd1:6800	http://localhost:6801
scrapyd2—scrapyd	http://scrapyd2:6800	http://localhost:6802
scrapyd3—scrapyd	http://scrapyd3:6800	http://localhost:6803
es—Elasticsearch API	http://es:9200	http://localhost:9200
spark—FTP	ftp://spark:21 & 30000-9	ftp://localhost:21 & 30000-9
Redis—Redis API	redis://redis:6379	redis://localhost:6379
MySQL - MySQL数据库	mysql://mysql:3306	mysql://localhost:3306

部分机器的`ssh`也是暴露的，Vagrant负责为我们重定向并转发这些端口，以避免冲突。我们所需要做的就是运行`vagrant ssh <hostname>`来访问想要连接的机器。

A.3　安装概述

我们所需安装的必要软件如下：

- Vagrant；
- git；
- VirtualBox（Windows或Mac主机）或Docker（Linux主机）。

在Windows中，可能还需要启用`git ssh`客户端。你可以访问它们的网站，并遵照它们对你所使用的平台描述的步骤操作。在下面几节中，我们将尝试提供逐步指引方案，目前来说这些方法是有效的，不过它们肯定会在未来某个时间失效，因此也请随时关注其官方文档。

A.4　在Linux上安装

我们之所以首先介绍如何在Linux上安装系统是因为它是最简单的。我将以Ubuntu 14.04.3 LTS (Trusty Tahr)进行演示，不过该过程在其他分发版本中也会十分相似，当然分发版本越不常见，你就越能了解如何填补其中的差距。为了安装Vagrant，需要访问Vagrant的网站：https://www.vagrant.com/ ，并浏览其下载页。右键单击**Debian package, 64-bit version** 。复制链接地址，如图A.2所示。



图A.2

我们将使用终端安装Vagrant，因为这是最通用的方式，尽管可以在Ubuntu上通过几下单击达成相同目的。为了打开终端，需要单击屏幕左上角的Ubuntu图标来打开**Dash**。另一种方案是，按下**Windows**按键。然后输入`terminal`，并单击**Terminal**图标以打开它。

我们输入`wget`，并粘贴从Vagrant页面中得到的链接。几秒后，将会下载一个`.deb`文件。输入`sudo dpkg -I <name of the .deb file you just downloaded>`以安装文件。到这里为止，Vagrant已经被安装好了。

安装`git`只需要在终端中输入如下两行命令。

```
$ sudo apt-get update

$ sudo apt-get install git
```

现在，让我们来安装Docker。我们将按照`https://docs.docker.com/engine/ installation/ubuntulinux/`的指南进行安装。在终

端中，输入如下命令。

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80

--recv-keys 58118E89F3A912897C070ADBF76221572C52609D

$ echo "deb https://apt.dockerproject.org/repo ubuntu-trusty main" | sudo

tee /etc/apt/sources.list.d/docker.list

$ sudo apt-get update

$ sudo apt-get install docker-engine

$ sudo usermod -aG docker $(whoami)
```

我们登出并再重新登录以应用分组变化，此时，应该可以没有问题地使用`docker ps`命令了。现在，我们可以下载本书的代码，并享受本书内容。

```
$ git clone https://github.com/scalingexcellence/scrapybook.git

$ cd scrapybook

$ vagrant up --no-parallel
```

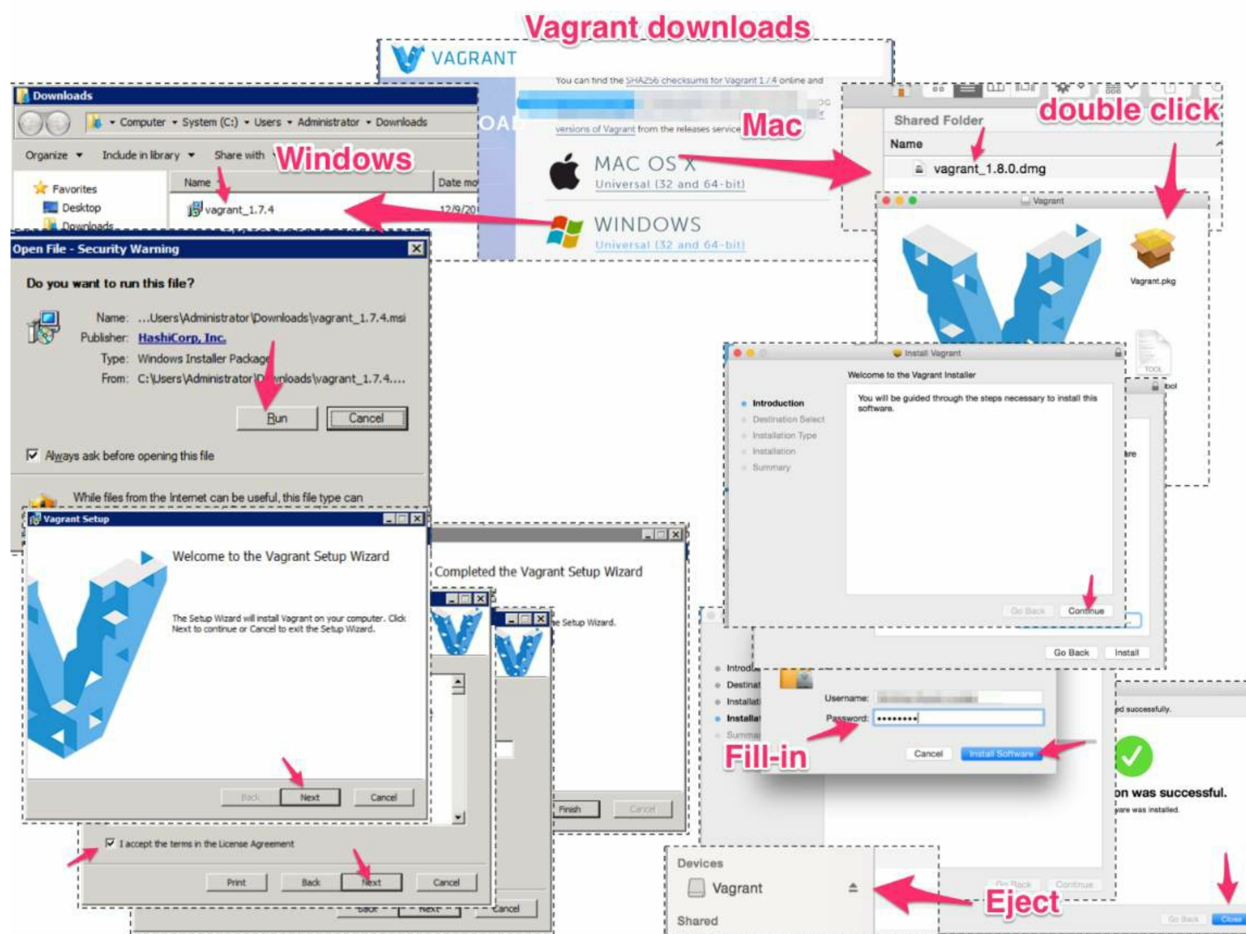
A.5 在Windows或Mac上安装

Windows和Mac环境中的安装过程是相似的，因此我们将一起介绍

这两种环境下的安装，并凸显它们之间的区别。

A.5.1　安装Vagrant

为了安装Vagrant，我们需要访问Vagrant的网站： https://www.vagrantup.com/ ，并浏览其下载页。选择自己的操作系统，并使用安装向导进行安装，如图A.3所示。

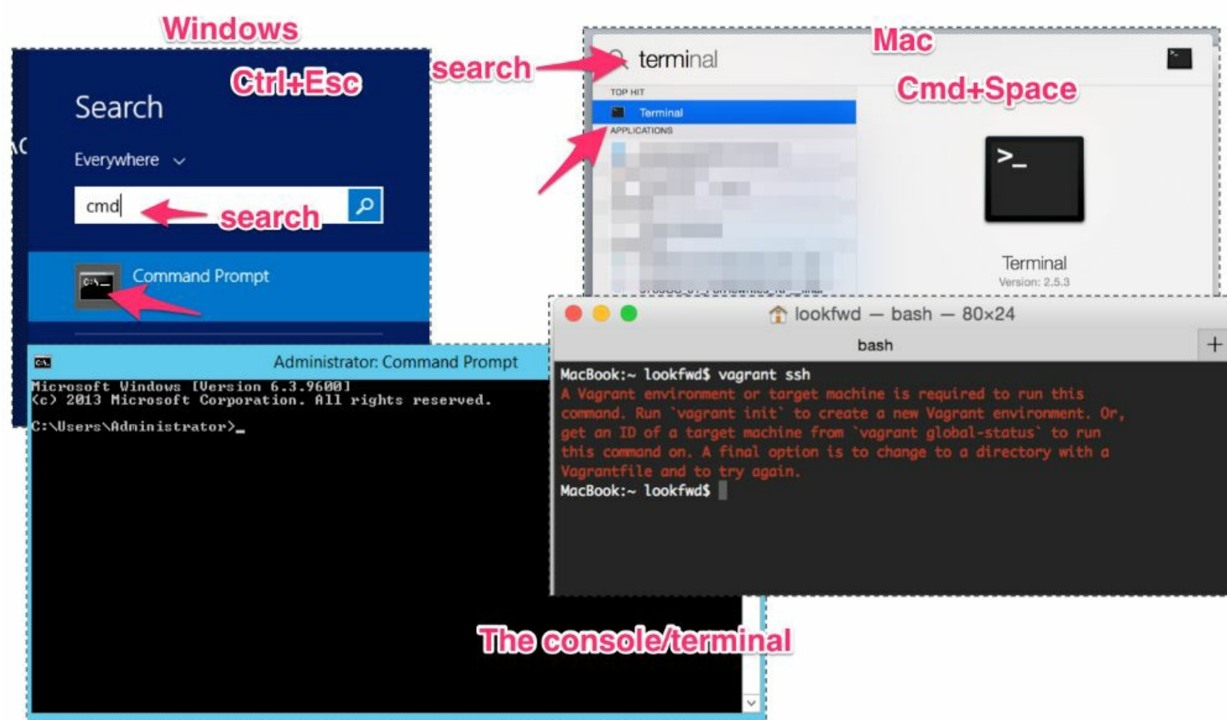


图A.3

几次单击之后，Vagrant将会安装好。要想访问它，需要打开命令行或终端。

A.5.2　如何访问终端

在Windows中，可以按下*Ctrl* + *Esc* 或*Win* 键打开应用菜单，并搜索`cmd`。而在Mac中，可以按下*Cmd* + *Space*，并搜索`terminal`。上述访问方式如图A.4所示。



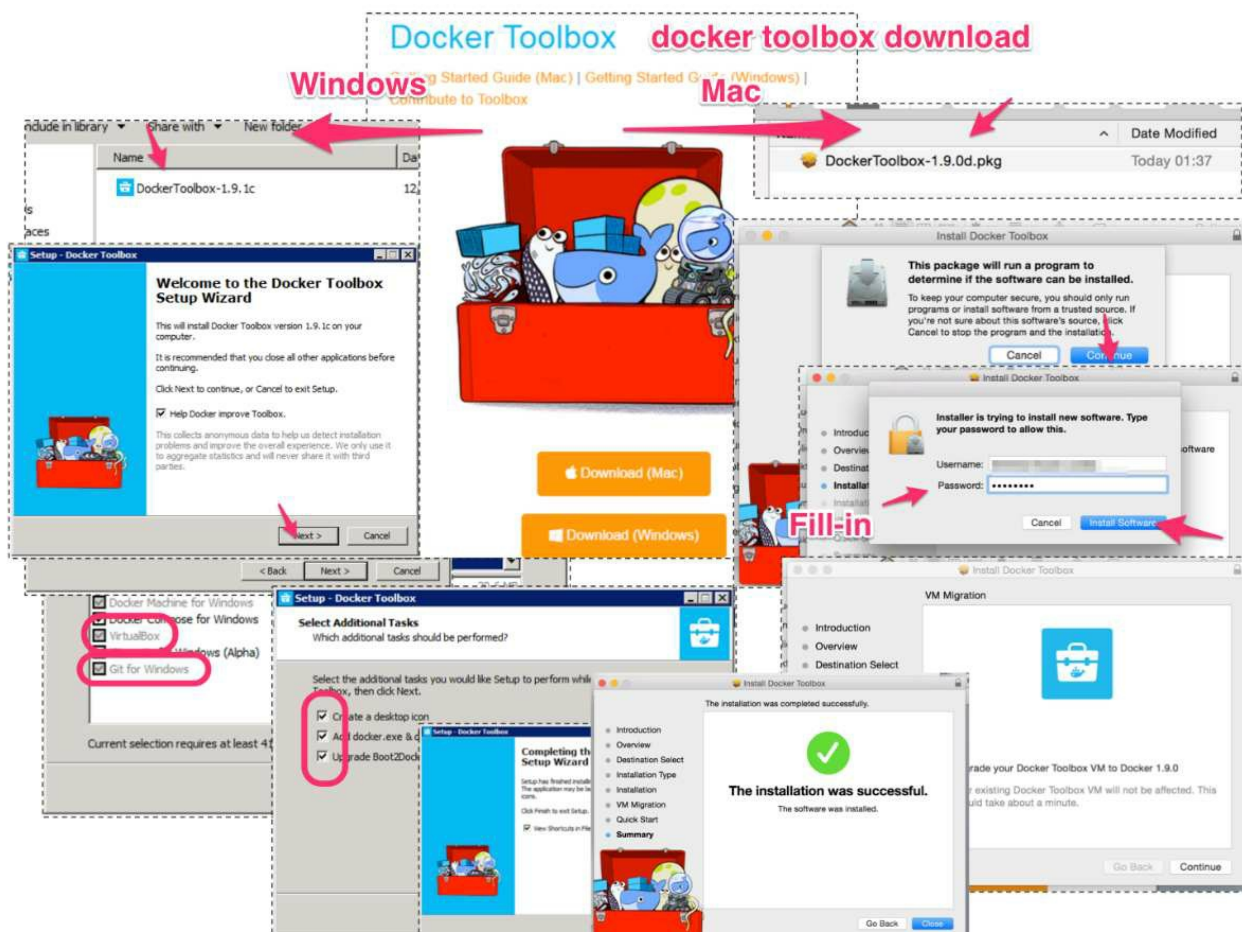
图A.4

无论哪种情况，我们都得到了一个控制台窗口，当我们输入`vagrant`时，将会打印出一些说明。这就是我们现在所需要做的所有事情。

A.5.3　安装VirtualBox和Git

为了简化该步骤，我们将安装Docker Toolbox，在其中已经包含了Git和VirtualBox。如果我们使用Google搜索*docker toolbox install*，可以

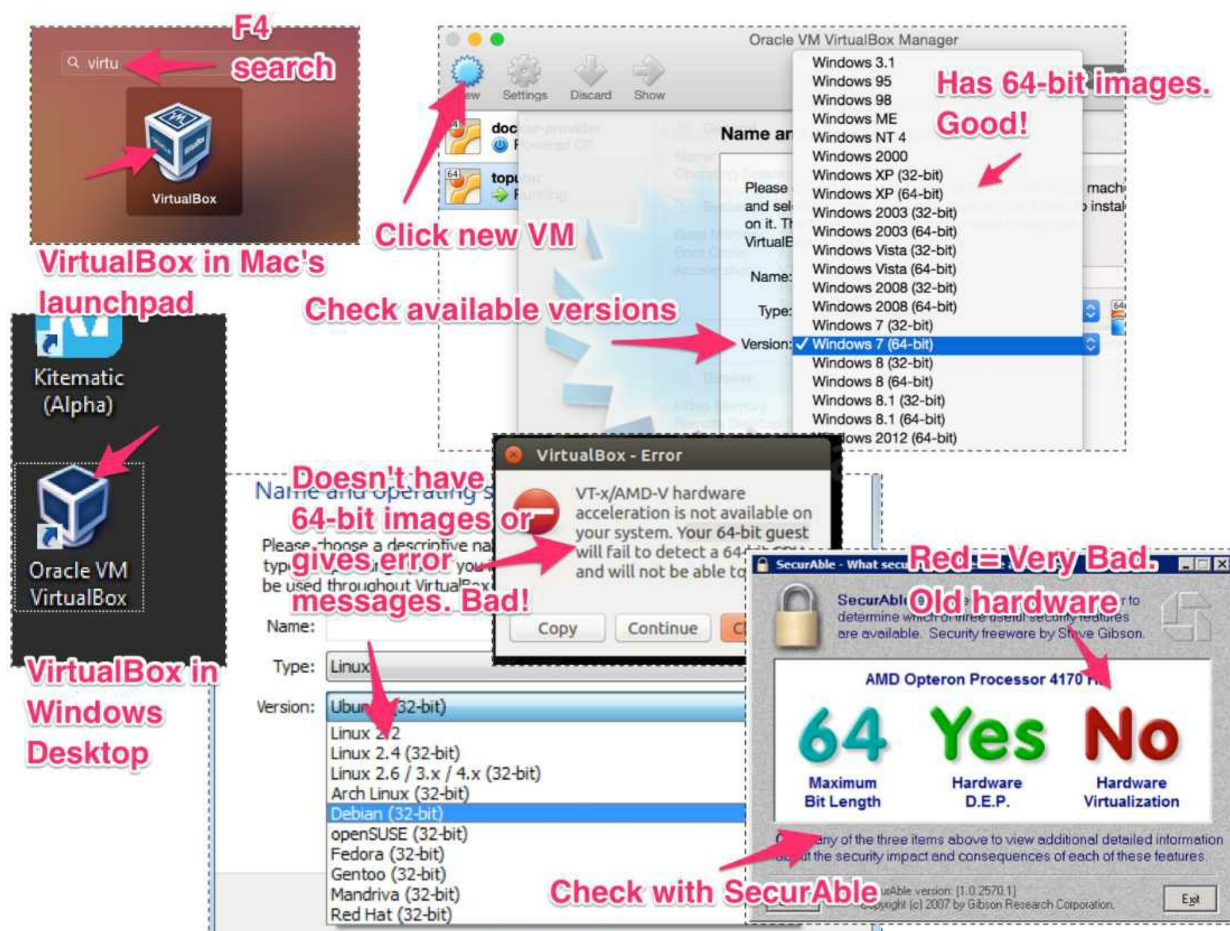
找到 https://www.docker.com/ docker-toolbox ，在这里可以下载适用于我们操作系统的版本。安装过程像Vagrant一样简单，如图A.5所示。



图A.5

A.5.4　确保VirtualBox支持64位镜像

安装好Docker Toolbox之后，可以在Windows桌面或Mac的启动器（按下F4打开）中找到VirtualBox的图标。尽早检查VirtualBox是否支持64位镜像非常重要，检查过程如图A.6所示。



图A.6

打开VirtualBox，单击**New** 按钮来创建一个新的虚拟机。查看版本下拉菜单，检查其中的选项，然后单击**Cancel** 。我们现在还不需要真正创建一个虚拟机。



如果下拉菜单中包含64位镜像，那么我们可以跳过本节接下来的部分。

如果下拉菜单中没有包含64位镜像，或者当我们尝试运行一个64位

虚拟机时得到类似**VT-x/AMD-V hardware acceleration is notavailable on your system** 的错误信息的话，我们可能就有一些麻烦了。

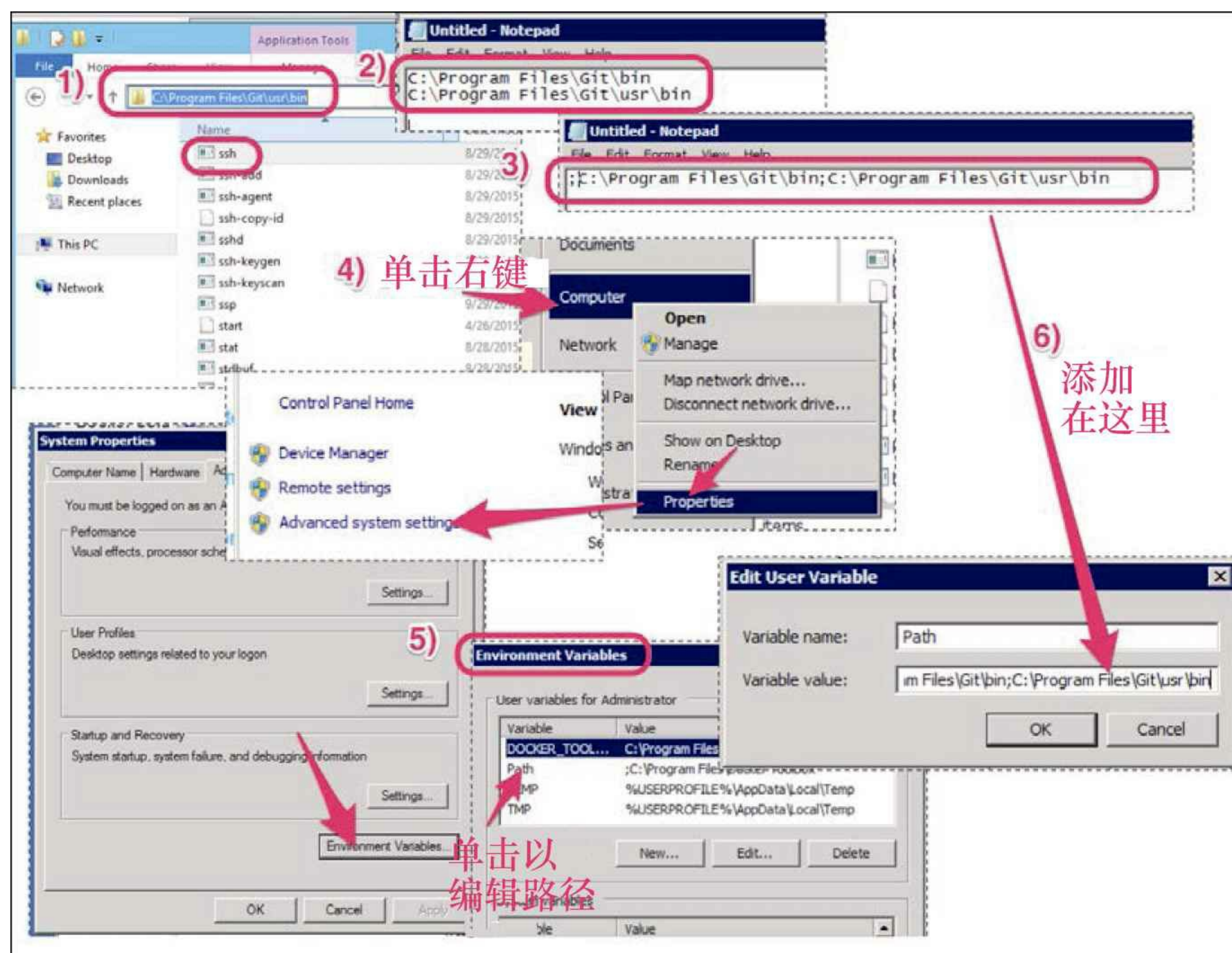
这意味着VirtualBox无法检测到我们电脑中的VT-x或AMD-V扩展。如果我们的硬件过旧，那么这种情况是合理且符合预期的。但是如果是新硬件，那么很可能是由于这些扩展在BIOS中被禁用了。如果我们使用的是Windows系统（很大可能），一个简单的方式是通过名为SecurAble的工具进行检查，该工具可以从 https://www.grc.com/securable.htm 中下载。如果**Hardware Virtualization** 为红色且提示为**No** 的话，就意味着我们的CPU不支持必要的虚拟扩展。在这种情况下，我们将无法运行Vagrant/Docker，不过我们仍然可以安装Scrapy，并且使用在线网站（`scrapybook.s3. amazonaws.com`）作为源来运行这些示例。我们可以从第4章中的爬虫开始使用，该爬虫是可以直接拿来使用的，并且是针对在线网站构建的。

如果**Hardware Virtualization** 为绿色，我们很可能可以从BIOS中启用该扩展。使用Google搜索你的电脑机型，以及如何变更BIOS中关于VT-x或AMD-V的设置。通常情况下，我们可以在重启时按下某个按键以访问BIOS。在这里，我们需要进入安全相关的菜单，然后启用**Virtualization Technology (VTx)** 或其他类似写法的选项。重启过后，我们将能够从该计算机运行64位的虚拟机。

A.5.5　在Windows中启用ssh客户端

如果我们使用的是Mac，将不需要本步，可以直接跳到下一节中。如果我们使用的是Windows，则没有提供给我们默认的`ssh` 客户端。幸运的是，Git（我们刚才安装的）有一个`ssh` 客户端，我们可以通过添加

Windows Path的方式激活它，如图A.7所示。



图A.7

默认情况下，ssh 的二进制文件位于C:\Program Files\Git\usr\bin 中（图A.7所示的1区域）。我们需要添加C:\Program Files\Git\usr\bin 和C:\Program Files\Git\bi``n 到路径当中。为了实现该目的，我们需要将它们复制到记事本中，并在每个路径前添加;来连接它们（如图A.7所示的3区域）。最终结果如下所示：

```
;C:\Program Files\Git\bin;C:\Program Files\Git\usr\bin
```

现在，按下*Ctrl* + *Esc* 或*Win* 按键，打开开始菜单，然后找到**Computer**（计算机） 选项。右键单击它（图A.7所示的4区域），并选择**Properties**（属性） 。在弹出的窗口中，选择**Advanced System Settings**（高级系统设置） 。然后，单击**Environment Variables**（环境变量） 。这里是我们用于编辑**Path** 的表单。单击**Path** 以编辑它。在**Edit User Variable**（编辑用户变量） 对话框中，我们在结尾处粘贴在记事本中连接的两个新路径。应当小心不要意外覆盖追加路径；之前的任何值。然后单击几次**OK**（确定） ，退出所有对话框，此刻必备软件已经全部安装完毕。

A.5.6　下载本书代码并创建系统

现在，我们已经拥有了一个功能齐全的Vagrant系统，接下来打开一个新的控制台/终端/命令行（我们已经在前面见过如何打开），输入如下命令，享受本书所带来的乐趣。

```
$ git clone https://github.com/scalingexcellence/scrapybook.git

$ cd scrapybook

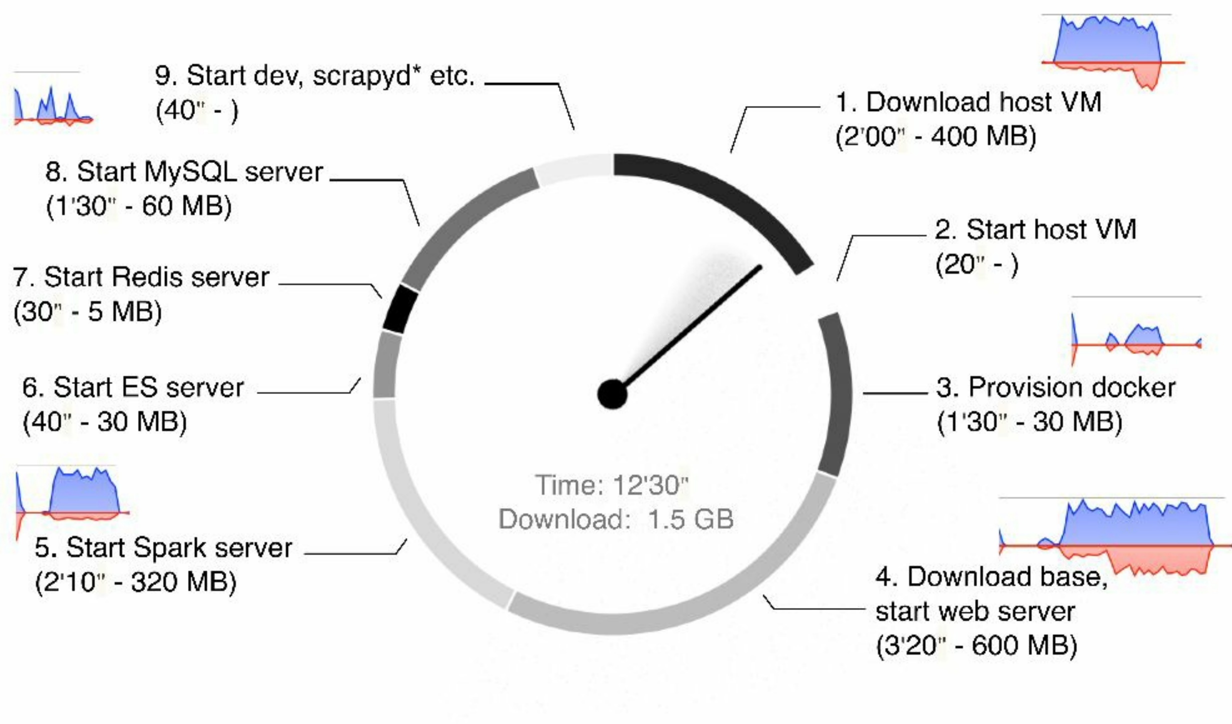
$ vagrant up --no-parallel
```

A.6　系统创建与操作FAQ

接下来是你在首次使用Scrapy工作时可能遇到的问题的解决方案。

A.6.1　我应该下载什么以及需要花费多少时间

当我们运行`vagrant up --no-parallel`之后，就没有那么多的可见度了。所经过的时间与我们的下载速度及网络连接质量密切相关。图A.8所示为当网络连接能力达到每秒下载5MB（38Mbit/s）内容时的期望时间。



图A.8

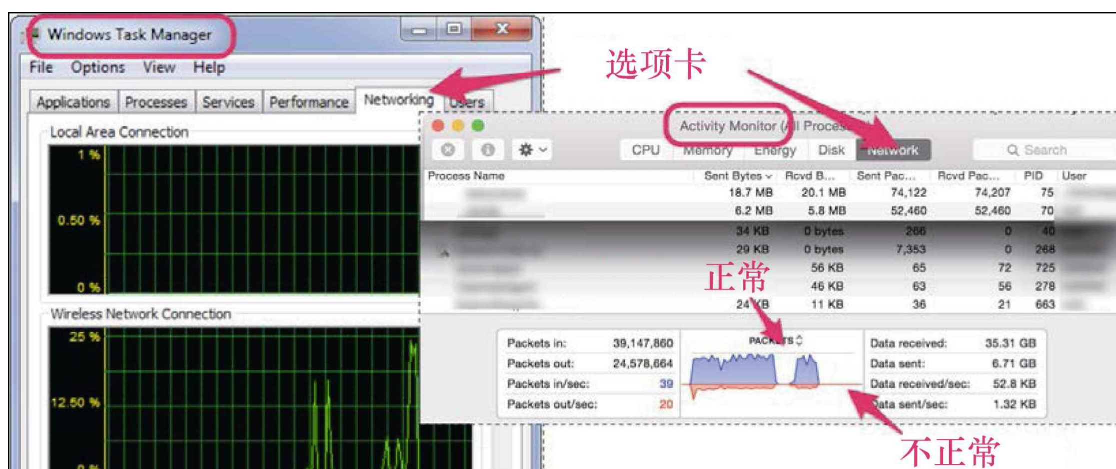
如果我们使用的是Linux环境，或是Docker已经被安装好，那么前三步就不是必要的，这样可以为我们节省4分钟的时间以及450MB的下载量。

请注意，上述所有步骤只与用于下载全部内容的`vagrant up --no-parallel`命令的第一次运行相关。后续运行在通常情况下只会花费不到10秒的时间。

A.6.2　如果Vagrant无法响应应该怎么办

可能会有很多原因导致Vagrant无法响应，我们所需要做的就是按下*Ctrl* + *C* 两次从中退出。然后再次尝试`vagrant up --no-parallel`，此时应当能够恢复。我们可能需要这样做几次，这取决于网络连接的

速度和质量。如果打开**Windows Task Manager**（**Windows**任务管理器） 或Mac的**Activity Monitor**（活动监视器），可以更清晰地看到Vagrant正在做什么，如图A.9所示。



图A.9

在下载期间或之后不超过60秒的短暂无法响应是可以预期的，因为此时软件正在进行安装。而更长时间的无法响应则很有可能意味着出现了某些问题。

当我们中断后再恢复时，`vagrant up --no-parallel` 可能会执行失败，并返回类似下面所述的错误信息。

```
Vagrant cannot forward the specified ports on this VM... The forwarded
port to 21 is already in use on the host machine.
```

这同样是一个临时性的问题。如果我们再次运行`vagrant up --no-parallel`，则应该能够成功恢复。

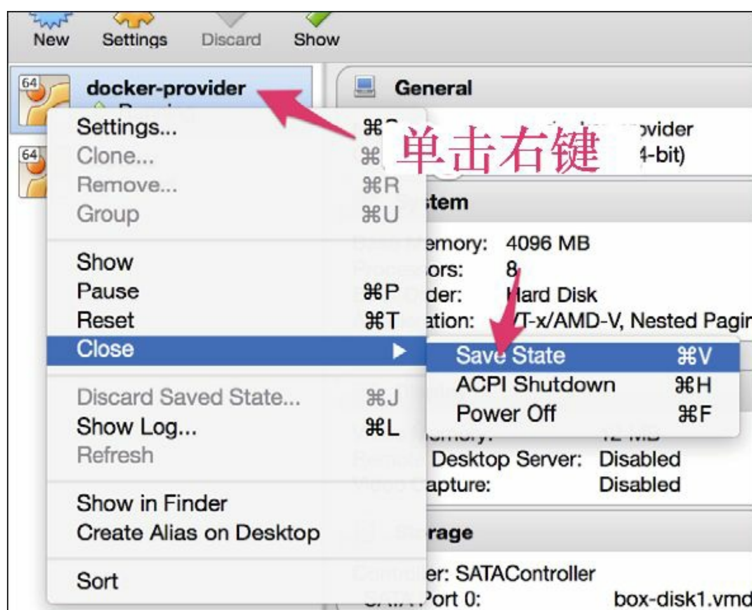
假设我们见到了如下的失败信息。

```
... Command: "docker" "ps" "-a" "-q" "--no-trunc"
Stderr: bash: line 2: docker: command not found
```

如果发生该情况，请按照下一个问题所显示的方法关闭并恢复虚拟机。

A.6.3　如何快速关闭/恢复虚拟机

当使用虚拟机时，最快的关闭方式是进入节能状态，具体来说就是打开VirtualBox，选择虚拟机，按下*Ctrl*+ *V* 或 *Cmd* + *V*，或右键单击菜单并选择**Save State**（保存状态），如图A.10所示。



图A.10

我们可以通过运行`vagrant up --no-parallel`恢复虚拟机。开

发和Spark服务器的`~/book`目录都应该可以正常工作。

A.6.4　如何完全重置虚拟机

如果我们想要变更核心数量、内存大小或虚拟机的端口映射，则需要进行完全重置。为了达到该目的，我们仍然需要按照前一个答案的步骤操作，不过现在要选择的是**Power Off**（关闭电源），或者按下*Ctrl* + *F* 或 *Cmd* + *F* 。我们也能通过编程方式完成此事，其执行语句是`vagrant global-status --prune`。我们可以找到名为“docker-provider”的虚拟主机的ID（比如95d1234），然后使用`vagrant halt`停止它，比如`vagrant halt 957d887`。

然后，可以使用`vagrant up --no-parallel`重启系统。不过很遗憾的是，开发和Spark机器很可能已经清空了其`~/book`目录。要想解决该问题，可以运行`vagrant destroy -f dev spark`，然后重新运行`vagrant up --no-parallel`。这将解决此类问题。

A.6.5　如何调整虚拟机大小

我们可能想要改变虚拟机的大小，比如将使用的内存从2GB调整为1GB，将使用的8核调整为4核。我们可以通过编辑`Vagrantfile.dockerhost`的`vb.memory`及`vb.cpus`设置来进行调整。然后，按照上一个答案的流程完全重置虚拟机。

A.6.6　如何解决端口冲突

有时，在主机上运行的一些服务可能占用了该系统需要的端口。首先，请注意如果我们打开了这两个机器的`Vagrantfile`，请移除其中

所有的`forwarded_port`语句，按照后面讲到的方法重置，此时仍然能够运行本书中的示例。我们可能刚好不太容易检查宿主机上这些端口运行的服务（通常通过Web浏览器）。

也就是说，我们可以通过重新映射冲突端口的方式更适当地解决冲突。让我们使用Web服务器9312端口的冲突作为示例。根据我们运行的是原生Linux还是虚拟机，过程会有些许不同。

Linux环境使用原生Docker

该问题将表现为如下所示的错误信息。

```
Stderr: Error: Cannot start container a22f...: failed to create
endpoint web on network bridge: Error starting userland proxy: listen
tcp 0.0.0.0:9312: bind: address already in use
```

打开Dockerfile，编辑Web服务器中`forwarded_port`语句的`host`值。之后，使用`vagrant destroy web`销毁Web服务器，并通过`vagrant up web`重启，如果问题发生在初始化加载阶段，则使用`vagrant up --no-parallel`恢复加载。

Windows或Mac环境使用虚拟机

此时，我们会得到不同的错误信息。

```
Vagrant cannot forward the specified ports on this VM, since they
would collide... The forwarded port to 9312 is already in use
on the host machine...
```

为了修复该问题，我们需要打开Vagrantfile.dockerhost，移除已有的包含端口号的行。然后在下面添加自定义端口转发语句，比如：config.vm.network “forwarded_port”, guest: 9312, host: 9316。此时将会修改为使用9316端口。接下来，按照“如何完全重置虚拟机”这一问题的答案流程重置虚拟机，一切又都会正常工作了。

A.6.7　如何隐藏在公司代理背后工作

有一些简单代理和TLS拦截代理。简单代理需要我们在请求到达互联网之前，转发到代理服务器上。它们可能需要权限验证，也可能不需要，不过无论哪种情况，我们需要使用的信息就是URL，该URL可以从我们的IT部门获取到。它大概形如http://user:pass@proxy.com:8080/。如果我们使用的是Linux，而不是虚拟机，很可能已经完全正确配置，不再需要进一步的调整。不过如果我们使用的是虚拟机，则需要使代理服务器在Vagrant、Docker provider VM、Ubuntu的APT下载以及Docker服务自身都应当可用。所有这些操作都已经在Vagrantfile.dockerhost中进行了处理，我们只需要移除定义proxy_url行的注释，并正确设置其值即可。

假设遇到了如下的SSL相关的问题。

```
SSL certificate problem: unable to get local issuer certificate
...
If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

无论是Vagrant还是部署的Docker，我们都很可能需要处理TLS拦截代理的问题。这种代理旨在以一种“中间人”的角色监控所有安全和不安全流量。它们代表我们执行https请求，在必要时验证证书；而我们执行到它们的https连接，验证它们的证书。我们的IT部门很可能会提供给我们一个证书，通常情况下是`.crt`文件的形式。我们将该文件的副本放到本书主目录下（`Vagrantfile`所在的目录）。接下来，按照前面例子设置`proxy_url`，然后更进一步取消掉定义`crt_filename`变量所在行的注释，将其值设置为我们的证书文件的名称。

A.6.8　如何连接Docker provider虚拟机

如果我们处于Linux环境中，并且没有使用虚拟机，那么我们的机器已经是Docker provider，此时无需做任何事情。如果我们使用的是虚拟机，那么可以通过运行`vagrant global-status --prune`得到Docker provider的ID，然后找到名为`docker-provider`的机器。我们可以在Linux或Mac环境中，使用别名的方式对其实现自动化。

```
$ alias provider_id="vagrant global-status --prune | grep 'docker-
provider' | awk '{print \$1}'"
```

我们可以使用`vagrant ssh <provider id>`，或者在已设置别名的情况下使用`vagrant ssh $(provider_id)`来连接Docker provider。在这里是Ubuntu Trusty 64位虚拟机。

A.6.9　每个服务器使用了多少CPU/内存

如果我们使用了原生Docker，或者按照前一个答案描述的方法连接到了provider，那么可以通过`docker stats`，看到每台独立Docker容器所消耗的资源，如下所示。

```
$ docker ps --format "{{.Names}}" | xargs docker stats
```

图A.11所示为运行第11章代码时的示例输出，此时是Scrapyd从Web服务器集中下载的时间。

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %
dev	0.11%	63.61 MB / 2.099 GB	3.03%
es	0.46%	295.1 MB / 2.099 GB	14.06%
mysql	0.09%	54.3 MB / 2.099 GB	2.59%
redis	0.06%	12.28 MB / 2.099 GB	0.59%
scrapyd1	128.36%	208.4 MB / 2.099 GB	9.93%
scrapyd2	118.59%	198.7 MB / 2.099 GB	9.47%
scrapyd3	114.12%	205.4 MB / 2.099 GB	9.79%
spark	1.17%	374.2 MB / 2.099 GB	17.83%
web	45.79%	79.84 MB / 2.099 GB	3.80%

图A.11

A.6.10　如何查看Docker容器镜像的大小

如果我们使用了原生Docker，或者按照之前答案中看到的方法连接到了provider，那么可以使用如下命令查看Docker镜像大小。

```
$ docker images
```

本书的容器都是基于一个镜像，每个变体上安装的其他软件都很少。因此，我们看到的GB级的大小是虚拟大小，而不是真实占用的磁盘空间。如果我们想要查看镜像的构建层次以及个体大小，可以为很长的dockviz 命令创建一个别名，然后按照如下所示进行使用。

```
$ alias dockviz="docker run --rm -v /var/run/docker.sock:/var/run/docker.
```

```
sock nate/dockviz"

$ dockviz images -t
```

A.6.11 当Vagrant无法响应时，如何重置系统

即使最终处于一个连Vagrant也无法重置的混乱状态，我们也可以对系统进行完全重置。我们可以在不重置虚拟主机的情况下做到这一点，当然这种方式需要花费一些时间来完成。我们所需要做的就是连接到docker provider机器，强行停止所有容器，移除它们的镜像，然后重启Docker。具体命令如下所示。

```
$ docker stop $(docker ps -a -q)

$ docker rm $(docker ps -a -q)
```

```
$ sudo service docker restart
```

也可以使用如下命令。

```
$ docker rmi $(docker images -a | grep "<none>" | awk "{print $3}")
```

我们使用这种方式移除了下载的所有Docker层内容，这就意味着下一次执行`vagrant up --no-parallel`时将会花费一些时间用于下载。

A.7　有一个无法解决的问题，怎么办

我们可以随时使用VirtualBox以及从osboxes.org（http://www.osboxes.org/ubuntu/ ）下载得到的Ubuntu 14.04.3（Trusty Tahr）镜像，按照Linux的安装过程操作。代码将会完全

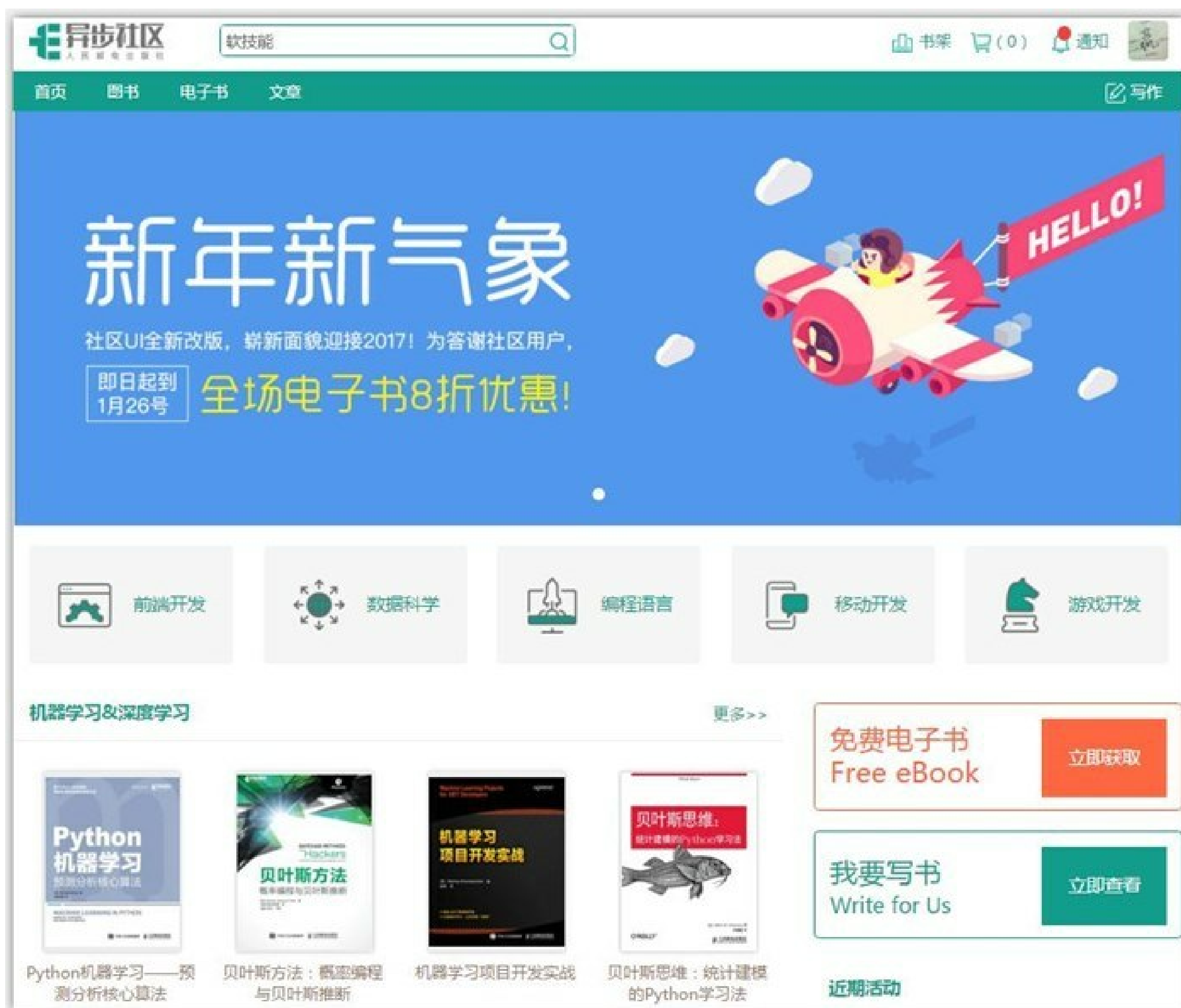
运行在虚拟机里。我们唯一会忽略的事情是端口转发和同步文件夹，这意味着要么我们手动设置它们，要么在虚拟机中进行开发。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。使用方法：注册成为社区用户，在下单购书时输入“**57AWG** ”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试

身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：436746675

社区网址： www.epubit.com.cn

官方微信： 异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn

O'REILLY

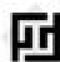
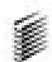
第2版



像计算机科学家一样思考Python

Think Python: How to Think Like a Computer Scientist, Second Edition

[美] Allen B. Downey 著
赵普明 译

 中国工信出版集团  人民邮电出版社
POSTS & TELECOM PRESS

目　　录

返回总目录

版权信息

书名：像计算机科学家一样思考Python（第2版）

ISBN：978-7-115-42551-5

- 著　[美] Allen B. Downey

  译　赵普明

  责任编辑　杨海玲

- 人民邮电出版社出版发行　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

本书以培养读者以计算机科学家一样的思维方式来理解Python语言编程。贯穿全书的主体是如何思考、设计、开发的方法，而具体的编程语言，只是提供了一个具体场景方便介绍的媒介。

全书共21章，详细介绍Python语言编程的方方面面。本书从最基本的编程概念开始讲起，包括语言的语法和语义，而且每个编程概念都有清晰的定义，引领读者循序渐进地学习变量、表达式、语句、函数和数据结构。书中还探讨了如何处理文件和数据库，如何理解对象、方法和面向对象编程，如何使用调试技巧来修正语法错误、运行时错误和语义错误。每一章都配有术语表和练习题，方便读者巩固所学的知识和技巧。此外，每一章都抽出一节来讲解如何调试程序。作者针对每章所专注的语言特性，或者相关的开发问题，总结了调试的方方面面。

本书的第2版与第1版相比，做了很多更新，将编程语言从Python 2升级成Python 3，并修改了很多示例和练习，增加了新的章节，更全面地介绍Python语言。

这是一本实用的学习指南，适合没有Python编程经验的程序员阅读，也适合高中或大学的学生、Python爱好者及需要了解编程基础的人阅读。对于第一次接触程序设计的人来说，是一本不可多得的佳作。

O’Reilly Media，Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数

百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

前言

本书的奇特历史

1999年，我正在为一门Java的编程入门课程备课。这门课我已经教过3个学期，感到有些灰心。课程的不及格率太高，即使是那些及格的学生，也只获得了很低的成就。

我发现问题之一是教材。它们太厚，有太多冗余的细节，而针对编程技巧的高阶的指导却很不足。而且学生们都有“陷阱效应”的苦恼：开头时很容易，也能循序渐进，但接着在第5章左右，整个地板就突然陷落了。新内容来得太多、太快，以至于我必须花费一学期剩下的全部时间来帮助他们拾回丢失的片段。

开课前两周，我决定自己来编写教材。我的目标有以下几个。

- 尽量简短。学生读10页书，比不读50页书要好。
- 注意词汇。我尝试尽量少用术语，并在第一次使用它们时做好定义。
- 循序渐进。为了避免陷阱效应，我抽出了最困难的课题，并把它们划分成更细的学习步骤。
- 专注于编程，而不是编程语言。我只注意包涵了Java的最小的可用子集，而忽略掉其他。

我需要一个标题，所以心血来潮选择了“How to Think Like a

Computer Scientist”。

第 1 版教材很粗糙，但确实有效。学生们读完课本，懂得了足够的基础知识，以至我甚至可以利用课堂时间和他们一起讨论更难、更有趣的话题，并且（最重要的是）可以让学生们有足够的时间在课堂上做练习。

我将这本书按照GNU自由文档许可协议（GNU Free Documentation License）发布，让用户可以复制、修改和分发本书。

接下来发生了最酷的事情。Jeff Elkner，弗吉尼亚州的一位高中老师，使用了我的书，并且将其翻译成Python语言的版本。他寄给我他的翻译副本，于是我有了一次很奇特的经历——通过读我自己的书来学习Python。通过绿茶出版社（Green Tea Press），在2001年我出版了第一个Python版本。

2003年，我开始在欧林学院（Olin College）教学，并第一次需要教授Python语言。和Java的对比非常惊人。学生们困扰更少，学会得更多，从事更有意思的项目，总的来说得到了更多的乐趣。

在那之后我一直继续拓展这本书的内容，修改错误，改进示例，并增加新的材料、尤其是练习。

结果就产生了本书，并改用了不那么宏伟堂皇的书名——*Think Python* 。部分改动如下所述。

- 我在每章的结尾添加了一节关于调试的说明。这些章节描述寻找和避免bug的通用技巧，并警示Python中容易出错的误区。

- 我增加了更多的练习，小到简短的理解性测试，大到几个实际工程。大部分练习都附带了链接，可以查看我的解答。
- 我添加了一系列案例研究——较长的示例，包括练习、解答以及讨论。
- 我扩展了关于程序开发计划和基础设计模式的讨论。
- 我增加了关于调试和算法分析的章节。

第2版增加了如下几个新特性。

- 全书内容和辅助代码都更新到Python 3。
- 增加了几节，以及更多关于Web的细节，以帮助初学者通过浏览器就能开始运行Python，而不需要过早地面对安装Python的问题。
- 对于第4章的“`turtle` 模块”，我把实现从以前自己开发的Swampy乌龟绘图包，改为使用更标准的Python模块`turtle`，它更容易安装，功能也更强大。
- 增加了新的一章“Python拾珍”（第19章），介绍Python提供的一些并不必需，但有时会很方便的特性。

我希望你喜欢这本书，并希望它至少能提供一点帮助，助你学会像计算机科学家那样编程和思考。

——Allen B. Downey

欧林学院

本书排版约定

本书使用下列排版约定。

- 中文楷体（英文斜体）：用于新术语、文件名和文件扩展名。
- 黑体字：表示术语表中定义的词汇。
- 等宽字体（`constant width`）：用于程序清单，以及段落中间的代码元素，如变量、函数名、数据库、数据类型、环境变量、语句或关键字等。
- 加粗等宽字体（**`constant with bold`**）：表示命令或其他应当由用户键入的文本。
- 等宽斜体字（*`constant width`*）：用于显示需要替换为用户提供的值或由环境确定的值的文本。

代码示例的使用

补充材料（代码示例、练习等）可以从http://www.greenteapress.com/thinkpython2/code 下载。

本书的目的是帮你完成工作。一般来说，只要是本书提供的示例代码，你都可以用在自己的程序和文档中。如果你不是要复制大部分的代码，就不需要联系我们申请授权。例如，写一个程序，里面使用了本书中的几段代码，不需要申请授权。但销售或分发O’Reilly书籍的示例光盘则需要授权。回答问题中引用本书内容或示例代码，并不需要申请授权，但将本书中大量的代码引入你的产品文档则需要授权。

在引用本书内容时，我们并不强求但鼓励你注明出处。引用通常包括书名、作者、出版社和ISBN。例如：“*Think Python, 2nd Edition* by

Allen B. Downey (O’Reilly). Copyright 2016 Allen Downey, 978-1-4919-3936-9”。

如果你觉得自己对本书代码示例的使用超出了上述授权范围，可以随时联系我们：permissions@oreilly.com。

联系我们

请将关于本书的评论和问题发给出版商。

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书提供了专门的网页，上面有勘误表、示例，以及其他额外的信息，可以通过http://bit.ly/think-python_2E访问该网页。

如果想对本书进行评论或想问技术问题，请将邮件发到bookquestions@oreilly.com。

想了解更多关于我们的书籍、课程、会议，以及新闻等信息，请登录我们的网站：http://www.oreilly.com 。

我们的其他联系方式如下。

Facebook：http://facebook.com/oreilly

Twitter：http://twitter.com/oreillymedia

YouTube：http://www.youtube.com/oreillymedia

致谢

非常感谢Jeff Elkner，他将我的Java书翻译成Python，这使我我开始了这个项目，并向我介绍了Python语言，结果Python成为我最喜爱的编程语言。

还要感谢Chris Meyers，他在*How to Think Like a Computer Scientist*一书中贡献了好几节。

感谢自由软件基金会（Free Software Foundation）开发了GNU自由文档协议，让我和Jeff以及Chris的合作成为可能。感谢创用CC（Creative Commons）开发了我们现在使用的协议。

感谢Lulu，负责*How to Think Like a Computer Scientist* 的编辑。

感谢O’Reilly Media负责Think Python一书的编辑。

感谢所有参与了本书早期版本编写的学生，以及所有（下面列出

的）贡献者提供的修订和建议。

贡献者列表

在最近几年中，超过100名眼光犀利、思维敏捷的读者给我寄来了建议和修订。他们对这个项目的贡献和热情，对我是极大的帮助。

如果你有建议或者修订意见，请发邮件到 feedback@thinkpython.com。如果我根据你的回馈做出了修改，会将你加入贡献者列表中（除非你要求被隐藏）。

如果你给出错误出现的位置的部分语句，会让我更容易搜索。页码或者章节编号也可以，但并不那么容易处理。谢谢！

- Lloyd Hugh Allen对8.4节提出了修订建议。
- Yvon Boulianne对第5章提出了一个语义错误的修订建议。
- Fred Bremmer对2.1节提出了一个修订建议。
- Jonah Cohen编写了Perl脚本将本书的LaTeX源码转换成美丽的HTML。
- Michael Conlon提出了第2章的一个语法错误，并提出第1章的格式改进，并且他开启了对解释器的技术讨论。
- Benoit Girard寄来一个对5.6节的有趣的修订。
- Courtney Gleason 和 Katherine Smith 编写了`horsebet.py`，在本书的早期版本中作为一个案例研究。他们的程序现在可以在网站上找到。
- Lee Harr提交了很多修订建议，我们没有空间在这里一一列出，并

且他确实应当被列为本书的一位主要编辑。

- James Kaylin是一名使用本书的学生。他提交了许多修订。
- David Kershaw 修正了3.10节中错误的`catTwice` 函数。
- Eddie Lam提出了第1章、第2章和第3章的很多修订建议，他也修正了Makefie，这样第一次运行时会自动建立索引。他也帮助我们设置了一个版本管理方案。
- Man-Yong Lee 寄来了对2.4节中的示例代码的修订。
- David Mayo指出第1章中的单词“unconsciously”需要被修改为“subconsciously”。
- Chris McAloon 寄来了对3.9节和3.10节的一些修订。
- Matthew J. Moelter是本书的长期贡献者，提出了很多修订建议。
- Simon Dicon Montford报告了第3章中缺失的函数定义以及几个错别字。他也发现了第13章中的`increment` 函数的错误。
- John Ouzts 修正了第3章中“返回值”的定义。
- Kevin Parks对关于本书如何分布提出了有价值的评论和建议。
- David Pool 发来了第1章中术语表中的错别字，以及鼓励的赞美之言。
- Michael Schmitt寄来了关于文件和异常的章节的修订建议。
- Robin Shaw指出了13.1节中的一个错误，printTime函数在一个示例中没有定义就使用了。
- Paul Sleigh在第7章中找到一个错误，并发现了Jonah Cohen用于生成HTML的Perl脚本的bug。
- Craig T. Snydal在德鲁大学（Drew University）的一门课上试验这个课本，他提出了好几个有价值的建议和修订。
- Ian Thomas和他的学生们使用这本书作为编程课程的教材。他们第

一个尝试使用本书后半部分的章节，并且提出了许多勘误和建议。

- Keith Verheyden发来了第3章的一个修正。
- Peter Winstanley让我们知道了第3章的拉丁文中一个长期存在的错误。
- Chris Wrobel修正了文件I/O和异常一章的代码错误。
- Moshe Zadka对本书有不可估量的贡献。他编写了关于字典的一章的第1版草稿，并在本书的早期阶段持续提供指导。
- Christoph Zwerschke发来了几个勘误和教学法的建议，并解释了gleich和selbe的区别。
- James Mayer发送给我们非常多的拼写错误，包括贡献者列表中的两个错误。
- Hayden McAfee发现了两个示例之间潜在的冲突。
- Angel Arnal是翻译本书的西班牙语版本的国际团队的一员。他也发现了英文版中的几个错误。
- Tauhidul Hoque 和Lex Berezhny创建了第1章中的图表，并改进了很多其他图表。
- Dr. Michele Alzetta发现了第8章中的一个错误，并发来了一些有趣的教学法评论，以及关于斐波那契数列和Old Maid的建议。
- Andy Mitchell发现了第1章中的一个录入错误，以及第2章中一个错误的示例。
- Kalin Harvey对第7章的一个说明提供了建议，并发现了几个录入错误。
- Christopher P. Smith发现了几个录入错误，并帮助我们更新本书到Python 2.2。
- David Hutchins发现了前言中的一个错别字。

- Gregor Lingl在奥地利维也纳的一个高中教授Python。他正在翻译本书的德文版，并发现了第5章中的几个错误。
- Julie Peters发现了前言中的一个错别字。
- Florin Oprina发来一个`makeTime` 的改进，`printTime` 的一个修正，以及发现的一个重要的录入错误。
- D. J. Webre对第3章的一个说明提出了建议。
- Ken在第8、9、11章中发现了好几个错误。
- Ivo Wever在第5章发现一个录入错误，并对第3章中的一个说明提出了建议。
- Curtis Yanko对第2章中的一个描述提出了建议。
- Ben Logan发来许多发现的录入错误，并发现了翻译HTML的问题。
- Jason Armstrong发现了第2章中一个漏掉的词。
- Louis Cordier发现了第16章中有一个代码和文本不一致的地方。
- Brian Cain在第2章和第3章中提出了几个描述的改进建议。
- Rob Black发来了许多勘误，包括一些针对Python 2.2的修改。
- 巴黎中央理工大学的Jean-Philippe Rey发来了一些补丁，包括对Python 2.2的更新，以及其他一些细心的改进。
- 乔治华盛顿大学的Jason Mader提供了许多有用的建议和改正。
- Jan Gundtofte-Bruun提醒我们“a error”应改为“an error”。
- Abel David和Alexis Dinno 提醒我们“matrix”的复数形式是“matrices”而不是“matrixes”。这个错误在书中已经存在了多年，但两个姓名以同样的字母开头的读者同一天报告了它。真的很奇怪。
- Charles Thayer鼓励我们删除掉一些语句结尾的分号，并建议我们理

清“形参”和“实参”的使用。

- Roger Sperberg指出了第3章的一个逻辑错误。
- Sam Bull指出了第2章中一段令人困惑的描述。
- Andrew Cheung指出了两处“定义前先使用”的错误。
- C.Corey Capel发现缺了单词，以及第4章的一个录入错误。
- Alessandra 帮助我们理清了一些关于Turtle的困惑。
- Wim Champagne在字典示例中发现一个错误。
- Douglas Wright在弧度计算中发现了一个除法向下取整的错误。
- Jared Spindor发现了一处句尾的无用词。
- Lin Peiheng发来了许多很有用的建议。
- Ray Hagtvedt发来了两处错误和一处不是那么错的错误。
- Torsten Hübsch指出Sawmpy中的一处不一致。
- Inga Petuhhov修正了第14章中的一个示例。
- Arne Babenhauserheide发来了几个有用的勘误。
- Mark E. Casida非常善于发现重复的单词。
- Scott Tyler填上了一个缺失的“that”，并发来了一堆勘误。
- Gordon Shephard发来了几个勘误，每个都用单独的邮件。
- Andrew Turner发现了第8章中的一个错误。
- Adam Hobart修正了一个在弧度计算中除法向下取整的错误。
- Daryl Hammond和Sarah Zimmerman指出我过早提出了`math.pi` 。并且Zim发现了一个录入错误。
- George Sass在调试章节中发现了一个bug。
- Brian Bingham建议了练习11-10。
- Leah Engelbert-Fenton指出我用`tuple` 作为变量名称，这恰恰违反了我自己的建议。然后他发现了一堆录入错误以及一个“定义前先

使用”。

- Joe Funke发现了一个录入错误。
- Chao-chao Chen在斐波那契示例中发现了一个不一致处。
- Jeff Paine知道space和spam的区别。
- Lubos Pintes发来一个录入错误。
- Gregg Lind和Abigail Heithoff建议了练习14-4。
- Max Hailperin发来了许多勘误和建议。Max是非凡的*Concrete Abstractions*（Course Technology, 1998）一书的作者之一。在读完本书之后你可能会想要读那本书。
- Chotipat Pornavalai在一个错误信息中发现了一个错误。
- Stanislaw Antol寄来了一个很有用的建议列表。
- Eric Pashman对第4章到第11章发来了许多勘误。
- Miguel Azevedo发现了一些录入错误。
- Jianhua Liu发来了一长列勘误。
- Nick King发现了一个缺失单词。
- Martin Zuther发来了一长列建议。
- Adam Zimmerman发现了我举例的一个“实例”中的不一致处，以及其他一些错误。
- Ratnakar Tiwari建议加一个脚注说明什么是“退化”三角形。
- Anurag Goel提出了`is_abecedarian`的另一个解答，并发来其他一些勘误。他还知道如何拼写Jane Austen。
- Kelli Kratzer发现了一个录入错误。
- Mark Griffiths指出了第3章中的一个令人困惑的示例。
- Roydan Ongie发现了我的牛顿方法的一个错误。
- Patryk Wolowiec帮我解决了一个HTML版本的问题。

- Mark Chonofsky告诉我Python 3中的新关键字。
- Russell Coleman帮我修正了几何错误。
- Wei Huang发现了几处录入错误。
- Karen Barber发现了本书中最古老的录入错误。
- Nam Nguyen发现了一个录入错误，并指出我使用了装饰器模式但并没有用它的名字。
- Stéphane Morin发来了一些建议和勘误。
- Paul Stoop修改了一个`uses_only` 中的录入错误。
- Eric Bronner指出了关于操作符顺序的讨论中的一个困惑之处。
- Alexandros Gezerlis提交的建议的数量和质量都设置了一个新的标准。我们非常感谢他！
- Gray Thomas知道哪边是左哪边是右。
- Giovanni Escobar Sosa发来一长列的勘误和建议。
- Alix Etienne修正了一个URL。
- Kuang He发现一个录入错误。
- Daniel Neilson修正了一个关于操作符顺序的错误。
- Will McGinnis指出`polyline` 在两个地方定义的不同。
- Swarup Sahoo发现了一个缺失的分号。
- Frank Hecker指出一个练习不细致，并发现了几个坏链接。
- Animesh B帮助我清理了一个令人困惑的示例。
- Martin Caspersen发现了两处取整错误。
- Gregor Ulm发来一些勘误和建议。
- Dimitrios Tsirigkas建议我更清晰地描述一个练习。
- Carlos Tafur发送了一整页勘误和建议。
- Martin Nordsletten在一个练习解答中找到了一个bug。

- Lars O. D. Christensen找到了一个失效的引用。
- Victor Simeone 找到了一个录入错误。
- Sven Hoexter指出一个叫作`input` 的变量名覆盖了内置函数名。
- Viet Le找到了一个录入错误。
- Stephen Gregory指出Python 3中`cmp` 的问题。
- Matthew Shultz告知我一个失效链接。
- Lokesh Kumar Makani告知我几个失效链接，以及出错消息的改变。
- Ishwar Bhat修正了我对费马大定理的描述。
- Brian McGhie建议了一个更清晰的阐述。
- Andrea Zanella将本书翻译成意大利语，并发送了一些勘误。
- 非常感谢Melissa Lewis和Luciano Ramalho出色的评论，以及对本书第2版的建议。
- 感谢PythonAnywhere的Harry Percival帮助人们在浏览器中运行Python。
- Xavier Van Aubel对第2版做出了几个有用的修正。

第1章　程序之道

本书的目标是教会你像计算机科学家一样思考。这种思考方式综合了数学、工程学以及自然科学的一些最优秀的特性。计算机科学家与数学家类似，他们使用形式语言来描述理念（特别是计算）；与工程师类似，他们设计产品，将元件组装成系统，对不同的方案进行评估选择；与自然科学家类似，他们观察复杂系统的行为，构建科学假说，并检验其预测。

作为计算机科学家，最重要的技能就是问题求解 。问题求解是发现问题、创造性地思考解决方案以及清晰准确地表达解决方案的能力。实践证明，学习编程的过程，正是训练问题求解能力的绝佳机会。这也是本章标题用“程序之道”的原因。

一方面，你将学会编程，其本身就是一个非常有用的技能；另一方面，你可以使用编程作为工具，去达到更高的目标。随着本书的深入，那个目标会逐渐明晰。

1.1　什么是程序

程序 是指一组定义如何进行计算的指令的集合。这种计算可能是数学计算，如解方程组或者查找多项式的根，也可以是符号运算，如搜索和替换文档中的文本，或者图形相关的操作，如处理图像或播放视频。

在不同的编程语言中，程序的细节有所不同，但几乎所有编程语言中都会出现以下几类基本指令。

- 输入： 从键盘、文件或者其他设备中获取数据。
- 输出： 将数据显示到屏幕，保存到文件中，或者发送到网络上等。
- 数学： 进行基本数学操作，如加法或乘法。
- 条件执行： 检查某种条件的状态，并执行相应的代码。
- 重复： 重复执行某种动作，往往在重复中有一些变化。

信不信由你，这差不多就是全部了。你所遇到过的所有程序，无论多么复杂，都是由类似上面的这些指令组成的。所以我们可以把编程看作一个将大而复杂的任务分解为更小的子任务的过程，不断分解，直到任务简单到足以由上面的这些基本指令组合完成。

1.2　运行Python

Python入门的挑战之一在于你可能需要自己在电脑上安装Python及相关软件。如果你熟悉自己的操作系统，而且习惯于命令行界面，那么安装Python不是什么问题。但对于初学者来说，同时学习编程和系统管理命令两件事，有时候是非常痛苦的。

为了避免这个问题，我推荐你开始先在浏览器中运行Python，等熟悉了Python语言之后，我再向你介绍如何在电脑上安装Python。

用于运行Python的网站有不少。如果你已经找到一个喜欢的，就可以直接去用。如果没有，我推荐PythonAnywhere。我在

http://tinyurl.com/thinkpython2e上提供了详细的入门指导。

有两个版本的Python，分别为Python 2和Python 3。它们很类似，所以如果你学会了一个版本，也能很容易地切换到另一个版本。实际上，作为初学者，你会遇到的两者之间的区别非常少。本书是针对Python 3编写的，但我也会给出一些关于Python 2的注意事项。

Python解释器 是一个读取并执行Python代码的程序。根据所在环境的不同，你可能需要点击程序图标，或者在命令行中键入`python` 命令来启动解释器。当它启动以后，可以看到如下输出：

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

前3行文本包含了解释器和所运行的操作系统的信息，所以可能与你看到的有些区别。但你应当检查版本号是否以3开头（本例所示的是`3.4.0` ），表示你使用的是Python 3的解释器。如果版本号以2开头，那么（你肯定猜到了）解释器是Python 2。

最后一行是一个提示符 ，表明解释器已经准备好，等待你键入代码。如果你键入一行代码并按下Enter键，解释器会显示结果：

```
>>> 1 + 1
2
```

1.3 第一个程序

依照传统，用新语言编写的第一个程序叫“Hello, World!”，因为这个程序所做的事情就是只显示“Hello, World!”。在Python中，它是这个样子：

```
>>> print('Hello, World!')
```

这是 **print** 语句 的一个示例。print 并不会真往纸上打印文字，而是在屏幕上显示结果。在这个例子中，输出的结果是：

```
Hello, World!
```

程序中的引号表示要显示的文本的开始和结束，在输出结果中它们并不显示。

括号表示print 是一个函数。我们将在第3章中讨论函数。

在Python 2中，print 语句略有不同。它不是一个函数，所以不使用括号：

```
>>> print 'Hello, World!'
```

这个区别的意义在后面会慢慢显现，但现在只需要知晓就足够了。

1.4　算术操作符

介绍完“Hello, World”之后，接下来是算术操作。Python提供了操作符，即像加号或减号这样的用来表达计算操作的特殊符号。

操作符+、-和*分别表示进行加法、减法和乘法运算，如下面示例所示：

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

操作符/表示除法运算：

```
>>> 84 / 2
42.0
```

这里你可能会奇怪为什么结果是`42.0`而不是`42`。我会在下一节解释。

最后，操作符** 表示进行指数运算。也就是说，会把一个数按指数进行乘方：

```
>>> 6**2 + 6
42
```

在其他一些语言中，指数操作用^ 符号表示，但在Python中^ 这个符号已经用来表示二进制按位运算XOR了。如果你不熟悉按位运算，结果可能会让你感到奇怪：

```
>>> 6 ^ 2
4
```

本书我不会讨论按位操作符，但读者可以在http://wiki.python.org/moin/BitwiseOperator上阅读相关文档。

1.5　值和类型

值（value）是程序操作的最基本的东西，如一个字母或者数字。前面我们见过一些值，如2 、42.0 以及'Hello，World!' 。

这些值属于不同的类型（type）：2 是整型（integer）的，42.0 是浮点型（floating-point）的，而'Hello，World!' 是字符串（string）类型的，这么称呼是因为它是由一堆字母“串连”起来的。

如果不确认一个值的类型，解释器可以告诉你：

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<type 'str'>
```

在这些结果中，单词“class”（类）被用于某一类型中，这是一种值类型。

不足为奇，整数属于`'int'`类型，字符串属于`'str'`类型，而浮点数属于`'float'`类型。

那么`'2'`和`'42.0'`这样的值呢？它们看起来像是数字，但又使用字符串常用的引号括起来：

```
>>> type('2')
<type 'str'>
>>> type('42.0')
<type 'str'>
```

它们是字符串。

当输入一个很大的数字时，你可能会忍不住想在数字中间加上逗号，就像`1,000,000`这样。在Python中这并不是合法的整数，但它凑巧又是一个合法的表达式：

```
>>> 1,000,000
(1, 0, 0)
```

当然，这和我们预期的完全不同！Python把`1,000,000`解释成一个用逗号分隔的整数序列。关于这种序列在本书后面可以学到更多内容。

1.6 形式语言和自然语言

自然语言 是指人们所说的语言，如英语、西班牙语和法语。它们不是由人设计而来的（虽然人们会尝试加以语法限制），而是自然演化而来的。

形式语言 则是人们为了特殊用途设计的语言。例如，数学上使用的符号体系是一种特别擅于表示数字和符号之间关系的形式语言；化学家则使用另一种形式语言来表示分子的化学结构。而最重要的是：

编程语言是人们为了表达计算过程而设计出来的形式语言。

形式语言倾向于对语法 做出严格的限制。例如，3 + 3 = 6是语法正确的数学表达式，但3+ = 3$6则不是。$H_2O$是语法正确的化学方程式，而$_2Zz$则不是。

语法规则有两种，分别适用于记号 （token）和结构（structure）。记号是语言的基本元素，如词、数字和化学元素。3+ = 3$6的一个问题就是$在数学表达式中（至少就我所知）不是合法记号。相似地，$_2Zz$不

合法是因为并不存在缩写为Zz的化学元素。

第二种语法规则指定记号所组合的方式。数学等式3+ = 3不合法，因为虽然+和=是合法记号，但不能将它们连续放置。相似地，在化学表达式里，下标数字应该出现在元素名称之后，而不是之前。

“This is @ well-structured Engli$h sentence with invalid t*kens in it.”是一个结构良好，但包含非法记号的英语语句。“This sentence all valid tokens has, but invalid structure with.”这句话所有的记号都合法，但是语句结构不合法。

当你阅读英语的句子或形式语言的语句时，需要弄清句子的结构是什么（虽然在自然语言中这个过程是下意识完成的）。这个过程称为语法分析 。

虽然形式语言和自然语言有很多共同的特点——记号、结构、语法以及语义，但它们也有一些区别。

- 歧义性： 自然语言充满了歧义，人们通过上下文线索和其他信息来处理这些歧义。形式语言通常设计为几乎或者完全没有歧义，即不论上下文环境如何，任何表达式都只有一个含义。
- 冗余性： 为了弥补歧义，减少误解，自然语言采用大量的冗余。因此，自然语言往往很啰嗦。形式语言则相对不那么冗余，更加简洁。
- 字面性： 自然语言充满了习惯用语和比喻。例如，有人说，“硬币掉了”（The penny dropped [1] ），并不一定是硬币，也不一定是有什么掉了。形式语言则严格按照它的字面意思表达含义。

因为我们都说着自然语言长大，有时候很难适应形式语言。在某种意义上，形式语言和自然语言的区别与诗词和散文的区别类似，而且程度更甚。

- 诗词： 字词的使用，既考虑它们的音韵，也考虑到它们的意义，而整首诗合起来表达某种意境或情绪反应。歧义不仅常见，而且常常是刻意为之。
- 散文： 字词的意义更加重要，而且句子的结构也提供更多的意义。散文比诗词更容易分析，但仍然有不少歧义。
- 程序： 计算机程序的意义不含歧义，直接如字面所指。完全可以通过它的记号和结构理解其意义。

形式语言的密度远远大于自然语言，所以阅读起来需要花费更多的时间。还有，结构非常重要，所以直接自顶向下、从左至右的阅读顺序并不一定是最好的。相反，要试着学会在头脑中解析程序，辨别出记号并解析出结构。最后，细节很重要。在自然语言中常常可以忽略的小错误，如拼写错误或者标点符号错误，在形式语言中往往会造成很大的差别。

1.7　调试

程序是很容易出错的。因为某种古怪的原因，程序错误被称为**bug**，而查捕bug的过程称为调试（debugging）。

一个程序中可能出现3种类型的错误：语法错误、运行时错误和语义错误。对它们加以区分，可以更快地找到错误。

编程，特别是调试，有时候会引发强烈的情绪。如果你挣扎于一个困难的bug，可能会感觉到愤怒、沮丧以及窘迫。

有证据表明，人们会像对待人一样对待电脑。当电脑良好完成工作时，我们会把它们当作队友，而当它们难以控制、粗暴无礼的时候，我们会按照对待那些粗暴固执的人一样对待它们（*The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*，Reeves和Nass著）。

对这些反应行为有所准备，可能会帮助你更好地对待电脑。一种方法是把它当作你的雇员，它有一定的长处，如速度和精度，也有特定的弱点，如没有同情心和无法顾全大局。

你的任务是做一个好经理：设法扬长避短，并找到方法控制你的情绪去面对问题，而不是让你的反应影响工作效率。

学习调试可能会带来挫折感，但它是一个有价值的技能，并在编程之外还有很多用途。每章的结尾处都有一节类似于本节的关于调试技巧的讨论。希望它们能带来帮助！

1.8　术语表

问题求解（problem solving）：总结问题、寻找解决方案以及表达解决方案的过程。

高级语言（high-level language）：设计来方便人们读写的编程语言，如Python。

低级语言 （low-level language）：设计来方便计算机执行的编程语言，也被称为“机器语言”或“汇编语言”。

可移植性 （portability）：程序的一种属性：可以在多种类型的计算机上运行。

解释器 （interpreter）：一个读取其他程序并执行其内容的程序。

提示符 （prompt）：解释器显示的文字，提示用户已经准备好接收用户的输入。

程序 （program）：一系列代码指令的集合，指定一种运算。

print语句 （print statement）：一个指令，可以通知Python解释器在屏幕上显示一个值。

操作符 （operator）：一种特殊符号，用来表达加法、乘法或字符串拼接等简单运算。

值 （value）：程序操作的数据基本单位，如一个数字或一个字符串。

类型 （type）：值的类别。到目前为止我们已经见过的类型有整数（`int` ）、浮点数（`float` ）和字符串（`str` ）。

整型 （integer）：用来表示整数的类型。

浮点型 （floating-point）：用来表示带小数部分的数的类型。

字符串 （string）：用来表示一串字符的类型。

自然语言 （natural language）：自然演化而来的人们所说的语言。

形式语言 （formal language）：人们设计为某些特定目的（如表达数学概念或者计算机程序）设计的任何一种语言。所有编程语言都属于形式语言。

记号 （token）：程序的语法结构的最基本单位，类似于自然语言中的词。

语法 （syntax）：用于控制程序结构的规则。

语法分析 （parse）：检查程序并分析其语法结构。

bug：程序中的错误。

调试 （debugging）：发现和纠正bug的过程。

1.9　练习

练习 **1-1**

在计算机前阅读本书是一个好主意，因为你可以边看边试验书中的示例。

每当你试验新的语言特性时，应当试着故意犯错。例如，在“Hello，World!”程序中，如果少写一个引号，会发生什么？如果两个引号都不写，会怎么样？如果把`print`拼写错了，会如何？

这种试验会帮你记住所读的内容，也能帮你学会调试，因为这样能看到不同的出错消息代表着什么。现在故意犯错总比今后在编码中意外出错好。

1．在print 语句中，如果漏掉一个括号，或者两个都漏掉，会发生什么？

2．如果正尝试打印一个字符串，那么若漏掉一个或所有的引号，会发生什么？

3．可以使用一个负号来表示负数，如-2 。如果在数字之前放一个正号，会发生什么？如果是2++2 呢？

4．在数学标记里，前置0是没有问题的，如02 。在Python中也这么做会发生什么？

5．如果在两个值之间不放任何操作符，会发生什么？

练习**1-2**

启动Python解释器，把它当作计算器使用。

1．在42分42秒中，一共有多少秒？

2．10千米相当于多少英里？提示：1英里相当于1.61千米。

3．如果你用42分42秒跑完10千米，那么你的平均速度（跑1千米需要的分钟和秒数）是多少？平均速度是多少千米每小时？

[1] “The penny dropped”在英语里的意思是：经过一段困惑后，突然理解了某个事情。——译者注

第2章　变量、表达式和语句

编程语言最强大的特性之一是操纵变量 的能力。变量是指向一个值的名称。

2.1　赋值语句

赋值语句 可以建立新的变量，并给它们赋值：

```
>>> message = 'And now for somthing completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

这个例子有3个赋值。第一个将一个字符串赋给叫作`message` 的变量；第二个将`17` 赋值给`n` ；第三个将π的（近似）值赋给变量`pi` 。

在纸上表达变量的一个常见方式是写下名称，并用箭头指向其值。这种图称为状态图 ，因为它显示了每个变量所在的状态 （请将它看作变量的心理状态）。图2-1显示了前面例子的状态图。

```
message ——> 'And now for something completely different'
      n ——> 17
     pi ——> 3.1415926535897932
```

图2-1　状态图

2.2　变量名称

程序员常常选择有意义的名称作为变量名——以此标记变量的用途。

变量名可以任意长短。它可以包含字母和数字，但必须以一个字母开头。使用大写字母是合法的，但变量名使用小写字母开头是个好主意（后面你会看到为何如此）。

下划线“_”可以出现在变量名称中。它经常出现在由多个词组成的变量名中，如`your_name`或`airspeed_of_unladen_swallow`。

如果给变量取非法的名称，会得到一个语法错误：

```
>>> 76trombones = 'big  parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones`非法，因为它以数字开头。`more@`非法，是因为它包含了一个非法字符`@`。但`class`有什么问题？

原因是`class`是Python的一个关键字。解释器通过关键字来识别程序的结构，并且它们不能用来作为变量名称。

Python 2共有31个关键字：

```
False       class       finally        is        return
None        continue     for          lambda      try
True         def         from         nonlocal  while
and          del         global       not       with
as           elif        if           or        yield
assert       else        import       pass
break        except      in           raise
```

你并不需要记住这个清单。在大多数开发环境中，关键字会以不同的颜色显示。如果把它们当作变量来用，会很容易发现。

2.3 表达式和语句

表达式 是值、变量和操作符的组合。单独一个值也被看作一个表达式，单独的变量也是如此。所以下面都是合法的表达式：

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

当你在提示符之后键入一个表达式时，解释器会对其进行求值，即尝试找到该表达式的最终值。在本例中，变量n 的值是17，而表达式n + 25 的值是42。

语句 是一段会产生效果的代码单元，如创建新变量或者显示一个值。

```
>>> n = 17
>>> print(n)
```

第一行是一个赋值语句，将值`17` 赋给变量`n` 。第二行是一个`print` 语句，显示变量`n` 的值。

当键入一行语句之后，解释器会执行它，也就是说会按照语句所说的来做。通常来说，语句本身没有值。

2.4　脚本模式

到目前为止我们都是在交互模式 （interactive mode）下运行Python，直接与解释器打交道。交互模式非常适合入门，但是，如果你需要编写超过几行的代码，它可能显得有点儿笨拙。

另一种编程模式是把代码保存称为脚本 的文件中，并以脚本模式（script mode）运行解释器，执行脚本。依照惯例，Python脚本文件通常以`.py` 结尾。

如果你已经了解在自己的电脑上如何创建和运行脚本，就可以继续学习了。否则我再次建议使用PythonAnywhere。我在http://tinyurl.com/thinkpython2e上写下了如何在脚本模式下运行的指导。

由于Python提供了两种运行模式，你可以在交互模式中尝试代码片段，然后将其放到脚本中。但交互模式和脚本模式还是有一些区别的，可能会引起困惑。

例如，如果使用Python作为计算器，你可能会输入：

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

第一行给变量`miles`赋值，但没有可见的效果。第二行是一个表达式，所以解释器对其进行求值，并显示结果。于是我们知道马拉松的长度大概是42千米。

但如果将上面同样的代码写入到脚本中并运行，则得不到任何输出。在脚本模式中，一个单独的表达式，也是没有可见效果的。Python实际上会对表达式进行求值，但不会显示其结果。除非你叫它这么做：

```
miles = 26.2
print (miles * 1.61)
```

这种现象一开始可能会让人迷惑。

脚本通常包含一系列的语句。如果语句超过一行，那么会随着语句执行的顺序一行行显示结果。

例如，脚本

```
print(1)
x = 2
print(x)
```

产生如下结果

```
1
2
```

赋值语句不会产生任何输出。

为了验证你的理解，可以在Python解释器中输入下面的语句，看它们做了什么：

```
5
x = 5
x + 1
```

现在把同样的语句存入到一个脚本文件并运行。输出是什么？修改脚本，将所有的表达式都转换成`print` 语句，再运行一遍。

2.5　操作顺序

当一个表达式中出现多个操作符时，求值的顺序依赖于优先级规则。对数学操作符，Python遵守数学的传统规则。缩略词**PEMDAS** 可以帮助记忆这些规则：

- 括号（**P**，Parentheses）拥有最高的优先级，并可以用来强制表达式按照你需要的顺序进行求值。因为括号中的表达式会先执行，所以`2*(3-1)` 的结果是4，而`(1+1)**(5-2)` 的结果是8。你也可以利用括号使得表达式更加易读，就像`(minute*100)/60` 这样，即使这里增加括号并不会改变结果。
- 乘方（**E**，Exponentiation）操作拥有次高的优先级，所以`1+2**3` 的结果是9，而不是27，而且`2 * 3**2` 的结果是18，而不是36。
- 乘法（**M**，Multiplication）和除法（**D**，Division）优先级相同，并且高于亦有相同优先级的加法（**A**，Addition）和减法（**S**，Substraction）。所以`2*3-1` 是5，而不是4，并且`6+4/2` 是8，而不是5。
- 优先级相同的操作按照自左向右的顺序求值（除了乘方以外）。所以表达式`degrees/2*pi`，除法在乘法之前执行，结果乘以`pi`。如果想除以2π，可以使用括号，或者写为`degrees/2/pi`。

其他操作符的优先级，我并不会花太多功夫记下来。如果只看表达式不能确定的话，使用括号指明优先级即可。

2.6 字符串操作

通常来说，字符串不能进行数学操作。即使看起来像数字也不行。下面的操作是非法的：

```
'2' - '1'     'eggs'/'easy'    'third'*'a charm'
```

但有两个例外：+和*。

操作符+进行字符串拼接（string concatenation）操作，意即将前后两个字符首尾连接起来。例如：

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

操作符*也适用于字符串；它进行重复操作。例如，'Spam'*3 的结果是'SpamSpamSpam' 。如果*的两个操作对象之一是字符串，那另一个必须是整数。’

字符串的+和*的应用，实际上和数字的加法与乘法类似。就像4*3 与4+4+4 相等一样，我们预期 'Spam'*3 与'Spam'+'Spam'+'Spam' 也相等，实际也确实如此。另一方面，字符串的拼接与重复操作和整数的加法与乘法操作也有很大的不同。你能够想出加法的一个属性，字符串拼接操作并不支持吗？

2.7 注释

当程序变得更大更复杂时，读起来也更困难。形式语言很紧凑，经常会遇到一段代码，却很难弄清它在做什么、为什么那么做。

因此，在程序中加入自然语言的笔记来解释程序在做什么，是个好主意。这种笔记被称为注释 （comments），它们以# 开头：

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

在这个例子里，注释单独占据一行。也可以把注释放到代码行的结尾：

```
percentage = (minute * 100) / 60     # percentage of an hour
```

从#开始到行尾的注释内容都会被解释器忽略掉——它们对程序本身运行没有任何影响。

注释最重要的用途在于解释代码并不显而易见的特性。我们可以合理地认为读者可以看懂代码在做什么，因此使用注释来解释为什么 这么做，要有用得多。

下面这段注释与代码重复，毫无用处：

```
v = 5         # 将5赋值给v
```

```
```

而下面这段注释则包含了代码中看不到的有用信息：

```
v = 5          # 速度，单位是米/秒
```

选择好的变量名称，可以减少注释的需要，但长名字也会让复杂表达式更难阅读，所以这两者之间需要衡量舍取。

2.8　调试

一个程序中可能出现3种错误：语法错误、运行时错误和语义错误。对它们加以区分，可以更快地找到错误。

语法错误

语法指的是程序的结构以及此结构的规则。例如，括号必须前后匹配，所以`(1+2)`是合法的，而`8)`就是一个语法错误。

程序中只要出现一处语法错误，Python就会显示出错消息并退出，你的程序就无法运行了。在编程生涯的最初几周中，可能会需要花费大量时间来查找语法错误。但随着经验的增加，犯错会越来越少，查找起来也会越来越快。

运行时错误

第二类错误是运行时错误，这样称呼是因为这种错误只有程序运行后才会出现。这些错误也常被称为异常 （exception），因为它们常常表示某些异常的（而且不好的）事情发生了。

运行时错误在开头几章中的简单示例里很少会出现，所以可能要过一段时间你才会遇到。

语义错误

第三类错误是语义错误，意思是错误与含义相关。如果你的程序中有一个语义错误，程序仍会成功运行，而不会产生任何出错消息，但是它不会执行正确的逻辑。它会做其他的事情。特别需要注意的是，它所做的正是你的代码所告诉它的。

查找语义错误会比较麻烦，因为需要反向查找，查看程序输出并尝试弄明白它到底做了什么。

2.9　术语表

变量 （variable）：引用一个值的名字。

赋值语句 （assignment statement）：将一个值赋值给变量的语句。

状态图 （state diagram）：用来展示一些变量以及其值的图示。

关键字 （keyword）：编译器或解释器保留的词，用于解析程序；变量名不能使用关键字，如`if`，`def`，`while`等。

操作数 （operand）：操作符所操作的值。

表达式 （expression）：变量、操作符和值的组合，可以表示一个单独的结果值。

求值 （evaluate）：对表达式按照操作的顺序进行计算，求得其结果值。

语句 （statement）：表示一个命令或动作的一段代码。至今我们见过赋值语句和打印语句。

执行 （execute）：运行一条语句，看它说的是什么。

交互模式 （interactive mode）：使用Python解释器的一种方式，在提示符之后键入代码。

脚本模式 （script mode）：使用Python解释器的一种方式，从脚本中读入代码并运行它。

脚本 （script）：保存在文件中的程序。

操作顺序 （order of operations）：当表达式中有多个操作符和操作对象要求值时，用于指导求值顺序的规则。

拼接 （concatenate）:将两个操作数首尾相连。

注释 （comment）：代码中附加的注解信息，用于帮助其他程序员阅读代码，并不影响程序的运行。

语法错误（syntax error）：程序中的一种错误，导致它无法进行语法解析（因此也无法被解释器执行）。

异常（exception）：程序运行中发现的错误。

语义（semantics）：程序表达的含义。

语义错误（semantic error）：程序中的一种错误，导致程序所做的事情不是程序员设想的。

2.10 练习

练习**2-1**

重申上一张的建议，每当你学习新语言特性时，都应当在交互模式中进行尝试，并故意犯下错误，看会有哪些问题。

- 我们已经见过`n = 42`是合法的。那么`42 = n`呢？
- 那么`x = y = 1`呢？
- 有些语言中，每个语句都需要以分号（;）结尾。如果你在Python语句的结尾放一个分号，会有什么情况？
- 如果在语句结尾放的是句号呢？
- 在数学标记中，对于`x`乘以`y`，可以这么表达：`xy`。在Python中这样尝试会有什么结果？

练习**2-2**

把Python解释器当作计算器来进行练习。

1．半径为r的球体的体积是$(4/3)\pi r^3$。半径为5的球体体积是多少？

2．假设一本书的定价是24.95美元，但是书店打了40%的折扣（6折）。运费是一本3美元，每加一本加75美分。60本书的总价是多少？

3．如果我6:52时离开家，并以慢速（6分10秒/千米）跑1.6千米，接下来以4分30秒/千米的速度跑4.8千米，再以慢速跑1.6千米。请问我回家吃早餐是什么时候？

第3章　函数

在程序设计中，函数 是指用于进行某种计算的一系列语句的有名称的组合。定义一个函数时，需要指定函数的名称并写下一系列程序语句。之后，就可以使用名称来“调用”这个函数。

3.1　函数调用

前面我们已经见过函数调用 的一个例子：

```
>>> type(42)
<class 'int'>
```

这个函数的名称是`type`，括号中的表达式我们称之为函数的参数。这个函数调用的结果是求得参数的类型。

我们通常说函数“接收”参数，并“返回”结果。这个结果也称为返回值 （return value）。

Python提供了一些可将某个值从一种类型转换为另一种类型的函数。`int` 函数可以把任何可以转换为整型的值转换为整型；如果转换失败，则会报错：

```
>>> int('32')
32
```

```
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int 可以将浮点数转换为整数，但不会做四舍五入操作，而是直接舍弃小数部分。

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float 函数将整数和字符串转换为浮点数：

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

最后，str 函数将参数转换为字符串：

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2　数学函数

Python有一个数学计算模块，提供了大多数常用的数学函数。模块（module）是指包含一组相关的函数的文件。

要想使用模块中的函数，需要先使用`import`语句将它导入运行环境：

```
>>> import math
```

这个语句将会创建一个名为`math`的模块对象（module object）。如果显示这个对象，可以看到它的一些信息：

```
>>> math
<module 'math' (built-in)>
```

模块对象包含了该模块中定义的函数和变量。若要访问其中的一个函数，需要同时指定模块名称和函数名称，用一个句点（.）分隔。这个格式称为句点表示法（dot notation）。

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

上面第一个例子使用了math.log10 来计算以分贝为单位的信号/噪声比（假设signal_power 和noise_power 都已经事先定义好了）。math 模块也提供了log 函数，用来计算底为e 的自然对数。

第二个例子计算radians 的正弦值。这个变量名已经暗示了，sin 以及cos 、tan 等三角函数接受的参数是以弧度 （radians）为单位的。若要将角度转换为弧度，可以除以180再乘以π：

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

表达式math.pi 从math 模块中获得变量pi 。这个变量的值是π的浮点近似值，大约精确到15位数字。

如果了解三角函数，可以把上面的结果和2的平方根的一半进行比较：

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3　组合

到现在为止，我们已经分别了解了程序的基本元素——变量、表达式和语句，但还没有接触如何将它们有机地组合起来。

程序设计语言最有用的特性之一就是可以将各种小的构建块（building block）组合 起来。例如，函数的参数可以是任何类型的表达式，包括算术操作符：

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

甚至还包括函数调用：

```
x = math.exp(math.log(x+1))
```

基本上，在任何可以使用值的地方，都可以使用任意表达式，只有一个例外：赋值表达式的左边必须是变量名称，在左边放置任何其他的表达式都是语法错误（后面我们还会看到这条规则的例外情况）。

```
>>> minutes = hours * 60                # 正确
>>> hours * 60 = minutes                # 错误!
SyntaxError: can't assign to operator
```

3.4　添加新函数

至此，我们都只是在使用Python提供的函数，其实我们也可以自己添加新的函数。函数定义 指定新函数的名称，并提供一系列程序语句，当函数被调用时，这些语句会顺序运行。

下面是一个例子：

```
def print_lyrics():
    print（"I'm a lumberjack，and I'm okay."）
    print（"I sleep all night and I work all day."）
```

`def` 是关键字，表示接下来是一个函数定义。这个函数的名称是`print_lyrics` 。函数名称的书写规则和变量名称一样：字母、数字和下划线是合法的，但第一个字符不能是数字。关键字不能作为函数名，而且我们应尽量避免函数和变量同名。

函数名后的空括号表示它不接收任何参数。

函数定义的第一行称为函数头 （header），其他部分称为函数体（body）。函数头应该以冒号结束，函数体则应当整体缩进一级。依照惯例，缩进总是使用4个空格，函数体的代码语句行数不限。

本例中`print` 语句里的字符串使用双引号括起来。单引号和双引号的作用相同。大部分情况下，人们都使用单引号，只在本例中这样的特殊情况下才使用双引号。本例中的字符串里本身就存在单引号（这里的单引号作为缩略符号用）。

代码中所有的引号（包括双引号和单引号）都必须是“直引号”，通

常在键盘上的Enter键附近。而“斜引号”，在Python中是非法的。

如果在交互模式里输入函数定义，则解释器会输出省略号（...）提示用户当前的定义还没有结束：

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

想要结束这个函数的定义，需要输入一个空行。

定义一个函数会创建一个函数对象，其类型是`'function'`。

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

调用新创建的函数的方式，与调用内置函数是一样的：

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

定义好一个函数之后，就可以在其他函数中调用它。例如，若想重

复上面的歌词，我们可以写一个repeat_lyrics 函数：

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

然后可以调用repeat_lyrics：

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

当然，这首歌其实并不是这么唱的。

3.5 定义和使用

将前面一节的代码片段整合起来，整个程序就像下面这个样子：

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

这个程序包含两个函数定义：print_lyrics 和repeat_lyrics 。函数定义的执行方式和其他语句一样，不同的是执行后会创建函数对象。函数体里面的语句并不会立即运行，而是等到函数被调用时才执行。函数定义不会产生任何输出。

你可能已经猜到，必须先创建一个函数，才能运行它。换言之，函数定义必须在函数被调用之前先运行。

作为练习，将程序的最后一行移动到首行，于是函数调用会先于函数定义执行。运行程序并查看会有什么样的错误信息。

现在将函数调用那一行放回到末尾，并将函数print_lyrics 的定义移到函数repeat_lyrics 定义之后。这时候运行程序会发生什么？

3.6　执行流程

为了保证函数的定义先于其首次调用执行，需要知道程序中语句运行的顺序，即执行流程 。

执行总是从程序的第一行开始。语句按照从上到下的顺序逐一运行。

函数定义并不会改变程序的执行流程，但应注意函数体中的语句并不立即运行，而是等到函数被调用时运行。

函数调用可以看作程序运行流程中的一个迂回路径。遇到函数调用

时，并不会直接继续运行下一条语句，而是跳到函数体的第一行，继续运行完函数体的所有语句，再跳回到原来离开的地方。

这样看似简单，但马上你就会发现，函数体中可以调用其他函数。当程序流程运行到一个函数之中时，可能需要运行其他函数中的语句。而后，当运行那个函数中的语句时，又可能再需要调用运行另一个函数的语句！

幸好Python对于它运行到哪里有很好的记录，所以每个函数执行结束后，程序都能跳回到它离开的地方。直到执行到整个程序的结尾，才会结束程序。

总之，在阅读代码时，并不总应该按照代码书写顺序一行行阅读；有时候，按照程序执行的流程来阅读代码，理解的效果可能会更好。

3.7　形参和实参 [1]

前面说到的函数有些需要传入参数 [2] 。例如，当调用`math.sin`时，需要传入一个数字作为实参。有的函数需要多个实参：`math.pow`需要两个，分别是基数（base）和指数（exponent）。

在函数内部，实参会赋值给称为形参（parameter）的变量。下面的例子是一个函数的定义，接收一个实参：

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

这个函数在调用时会把实参的值赋到形参bruce 上，并将其打印两次。

这个函数对任何可以打印的值都可用。

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

内置函数的组合规则，在用户自定义函数上也同样可用，所以我们可以对print_twice 使用任何表达式作为实参：

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

作为实参的表达式会在函数调用之前先执行。所以在这个例子中，表达式'Spam '*4 和math.cos(math.pi) 都只执行一次。

也可以使用变量作为实参：

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

作为实参传入到函数的变量的名称（michael ）和函数定义里形参的名称（bruce ）没有关系。函数内部只关心形参的值，而不用关心它在调用前叫什么名字；在print_twice 函数内部，大家都叫bruce 。

3.8 变量和形参是局部的

在函数体内新建一个变量时，这个变量是局部的（local），即它只存在于这个函数之内。例如：

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

这个函数接收两个实参，将它们拼接起来，并将结果打印两遍。下面是一个使用这一函数的例子：

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

当cat_twice 结束时，变量cat 会被销毁。这时再尝试打印它的话，会得到一个异常：

```
>>> print(cat)
NameError: name 'cat' is not defined
```

形参也是局部的。例如，在print_twice 函数之外，不存在bruce这个变量。

3.9　栈图

要跟踪哪些变量在哪些地方使用，有时候画一个栈图 （stack diagram）会很方便。和状态图一样，栈图可以展示每个变量的值，不同的是它会展示每个变量所属的函数。

每个函数使用一个帧 包含，帧在栈图中就是一个带着函数名称的盒子，里面有函数的参数和变量。前面的函数示例的栈图如图3-1所示。

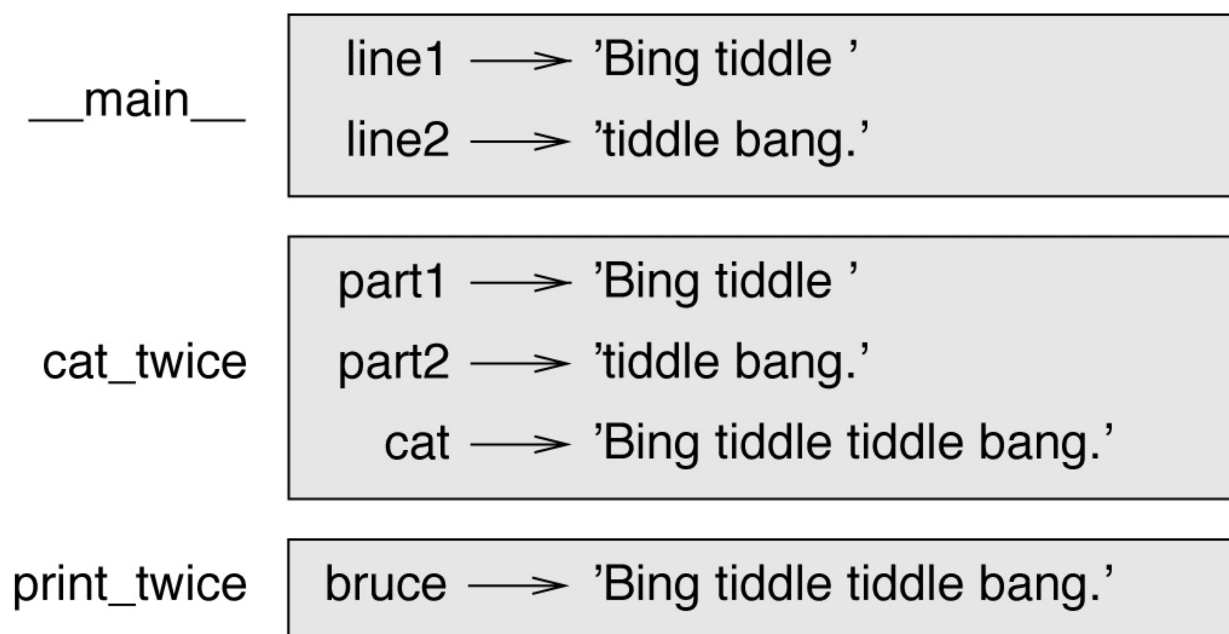


图3-1　栈图

图中各个帧从上到下安排成一个栈，能够展示出哪个函数被哪个函数调用了。在这个例子里，`print_twice` 被`cat_twice` 调用，而`cat_twice` 被`__main__` 调用。`__main__` 是用于表示整个栈图的图框的特别名称。在所有函数之外新建变量时，它就是属于`__main__` 的。

每个形参都指向与其对应的实参相同的值，所以，`part1` 和`line1` 的值相同，`part2` 和`line2` 的值相同，而`bruce` 和`cat` 的值相同。

如果调用函数的过程中发生了错误，Python会打印出函数名、调用它的函数的名称，以及调用这个调用者的函数名，依此类推，一直到`__main__` 。

例如，如果在`print_twice` 中访问`cat` 变量，则会得到一个`NameError:`

```
Traceback (innermost last):
    File "test.py", line 13, in __main__
        cat_twice(line1, line2)
    File "test.py", line 5, in cat_twice
        print_twice(cat)
    File "test.py", line 9, in print_twice
        print cat
NameError: name 'cat' is not defined
```

上面这个函数列表被称为回溯（traceback）。它告诉你错误出现在哪个程序文件，哪一行，以及哪些函数正在运行。它也会显示导致错误的那一行代码。

回溯中函数的顺序和栈图中图框的顺序一致。当前正在执行的函数在最底部。

3.10 有返回值函数和无返回值函数

在我们使用过的函数中，有一部分函数，如数学函数，会返回结果。因为没有想到更好的名字，我称这类函数为有返回值函数（fruitful function）。另一些函数，如`print_twice`，会执行一个动作，但不返回任何值。我们称这类函数为无返回值函数（void function）。

当调用一个有返回值的函数时，大部分情况下你都想要对结果做某种操作。例如，你可能会想把它赋值给一个变量，或者用在一个表达式中：

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

在交互模式中调用函数时，Python会直接显示结果：

```
>>> math.sqrt(5)
2.2360679774997898
```

但是在脚本中，如果只是直接调用这类函数，那么它的返回值就会永远丢失掉！

```
math.sqrt(5)
```

这个脚本计算5的平方根，但由于并没有把计算结果存储到某个变量中，或显示出来，所以其实没什么实际作用。

无返回值函数可能在屏幕上显示某些东西，或者有其他的效果，但是它们没有返回值。如果把该结果赋值给某个变量，则会得到一个特殊的值None 。

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

值None 和字符串'None' 并不一样。它是一个特殊的值，有自己独特的类型：

```
>>> print type(None)
<class 'NoneType'>
```

到目前为止，我们自定义的函数都是无返回值函数。再过几章我们就会开始写有返回值的函数了。

3.11 为什么要有函数

为什么要花功夫将程序拆分成函数呢？也许刚开始编程的时候这其中的原因并不明晰。下面这些解释都可作为参考。

- 新建一个函数，可以让你有机会给一组语句命名，这样可以让代码更易读和更易调试。
- 函数可以通过减少重复代码使程序更短小。后面如果需要修改代码，也只要修改一个地方即可。
- 将一长段程序拆分成几个函数后，可以对每一个函数单独进行调试，再将它们组装起来成为完整的产品。
- 一个设计良好的函数，可以在很多程序中使用。书写一次，调试一次，复用无穷。

3.12 调试

你将会掌握的一个最重要的技能就是调试。虽然调试可能时有烦恼，但它的确是编程活动中最耗脑力、最有挑战、最有趣的部分。

在某种程度上，调试和刑侦工作很像。你会面对一些线索，而且必须推导出事情发生的过程，以及导致现场结果的事件。

调试也像是一种实验科学。一旦猜出错误的可能原因，就可以修改程序，再运行一次。如果猜对了，那么程序的运行结果会符合预测，这样就离正确的程序更近了一步。如果猜错了，则需要重新思考。正如夏洛克·福尔摩斯所说的："当你排除掉所有的可能性，那么剩下的，不管多么不可能，必定是真相。"（柯南·道尔《四签名》）

对某些人来说，编程和调试是同一件事。也就是说，编程正是不断调试修改直到程序达到设计目的的过程。这种想法的要旨是，应该从一个能做某些事的程序开始，然后做一点点修改，并调试修改，如此迭代，以确保总是有一个可以运行的程序。

例如，Linux是包含了数百万行代码的操作系统，但最开始只是Linus Torvalds编写的用来研究Intel 80386芯片的简单程序。据Larry Greenfield所说："Linus最早的一个程序是交替打印AAAA和BBBB。后来这些程序演化成了Linux。"（《Linux用户指南》Beta版本1）

3.13　术语表

函数（function）：一个有名称的语句序列，可以进行某种有用的操作。函数可以接收或者不接收参数，可以返回或不返回结果。

函数定义 （function definition）：一个用来创建新函数的语句，指定函数的名称、参数以及它包含的语句序列。

函数对象 （function object）：函数定义所创建的值。函数名可以用作变量来引用一个函数对象。

函数头 （header）：函数定义的第一行。

函数体 （body）：函数定义内的语句序列。

形参 （parameter）：函数内使用的用来引用作为实参传入的值的名称。

函数调用 （function call）：运行一个函数的语句。它由函数名称和括号中的参数列表组成。

实参 （argument）：当函数调用时，提供给它的值。这个值会被赋值给对应的形参。

局部变量 （local variable）：函数内定义的变量。局部变量只能在函数体内使用。

返回值 （return value）：函数的结果。如果函数被当作表达式调用，返回值就是表达式的值。

有返回值函数 （fruitful function）：返回一个值的函数。

无返回值函数 （void function）：总是返回None 的函数。

None：由无返回值函数返回的一个特殊值。

模块 （module）：一个包含相关函数以及其他定义的集合的文件。

import语句 （import statement）：读入一个模块文件，并创建一个模块对象的语句。

模块对象 （module object）：使用`import`语句时创建的对象，提供对模块中定义的值的访问。

句点表示法 （dot notation）：调用另一个模块中的函数的语法，使用模块名加上一个句点符号，再加上函数名。

组合 （composition）：使用一个表达式作为更大的表达式的一部分，或者使用语句作为更大的语句的一部分。

执行流程 （flow of execution）：语句运行的顺序。

栈图 （stack diagram）：函数栈的图形表达形式，也展示它们的变量，以及这些变量引用的值。

图框 （frame）：栈图中的一个图框，表达一个函数调用。它包含了局部变量以及函数的参数。

回溯 （traceback）：当异常发生时，打印出正在执行的函数栈。

3.14　练习

练习**3-1**

编写一个函数`right_justify`，接收一个字符串形参`s`，并打印出足够的前导空白，以达到最后一个字符显示在第70列上。

```
>>> right_justify('monty')
                                                                      monty
```

提示：可以利用字符串的拼接和重复特性。另外，Python提供了一个内置名为`len`的函数，返回一个字符串的长度，所以`len('allen')`的值是5。

练习**3-2**

函数对象是一个值，可以将它赋值给变量，或者作为实参传递。例如，`do_twice`是一个函数，接收一个函数对象作为实参，并调用它两次：

```
def do_twice(f):
    f()
    f()
```

下面是一个使用`do_twice`来调用一个`print_spam`函数两次的示例：

```
def print_spam():
```

```
    print('spam')

do_twice(print_spam)
```

1．将这个示例存入脚本中并测试它。

2．修改do_twice，让它接收两个实参，一个是函数对象，另一个是一个值，它会调用函数对象两次，并传入那个值作为实参。

3．将本章前面介绍的函数print_twice的定义复制到你的脚本中。

4．使用修改版的do_twice来调用print_twice两次，并传入实参'spam'。

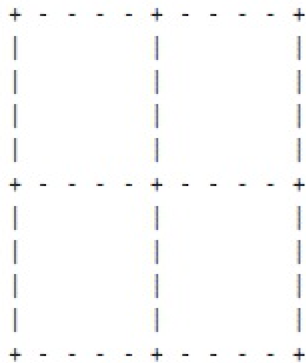
5．定义一个新的函数do_four，接收一个函数对象与一个值，使用这个值作为实参调用函数4次。这个函数的函数体应该只有2条语句，而不是4条。

解答：http://thinkpython2.com/code/do_four.py。

练习**3-3**

注意：这个练习应该只用语句和我们已经学过的其他语言特性实现。

1．编写一个函数，绘制如下的表格：



提示：要在同一行打印多个值，可以使用逗号分隔不同的值：

```
print('+', '-')
```

默认情况下，Print会自动换行，如果你想改变这一行为，在结尾打印一个空格，可以这样做：

```
print('+', end=' ')
print('-')
```

这两条语句的输出是'+ -'。

不带参数的print语句会结束当前行并开始下一行。

2．编写一个函数绘制类似的表格，但有4行4列。

解答：http://thinkpython2.com/code/grid.py。鸣谢：这个练习基于Oualline的《实践C编程》第3版（O'Reilly Media，1997）中的一个示

例。

---

[1] 这一段中讲的参数有两种：函数定义里的形参（parameter），以及调用函数时传入的实参（argument），这里两种是有区分的。——译者注

[2] 调用时传入的参数称为实参（argument）。——译者注

第4章　案例研究：接口设计

本章通过一个案例研究来展示设计互相配合的函数的过程。

本章介绍`turtle`模块，通过这个模块可以使用乌龟图形来创造图像。大多数Python安装包里都包含了`turtle`模块，但是如果使用的是PythonAnywhere，则不能直接运行乌龟示例（至少在我写本这本书的时候还不行）。

如果你已经在电脑上安装了Python，应该可以运行本章的示例。否则，现在就是安装Python的好时机。我在http://tinyurl.com/thinkpython2e上写了安装指南。

本章的代码示例可以从http://thinkpython2.com/code/polygon.py下载。

4.1　turtle模块

要检查是否已经安装了`turtle`模块，可以打开Python解释器并在其中输入：

```
>>> import turtle
>>> bob = turtle.Turtle()
```

运行这段代码时，应该会创建一个新窗口，里面有一个小箭头代表乌龟。请关闭窗口。

创建一个文件mypolygon.py，并输入如下代码：

```
import tutle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

turtle 模块（以小写t开头）提供一个叫作Turtle 的函数（以大写T开头），它会创建一个Turtle对象，我们将其赋值到bob 变量。打印bob 会得到类似下面的输出：

```
<turtle.Turtle object at 0xb7bfbf4c>
```

这意味着bob 变量引用着在turle 模块中定义的Turtle 类型的一个对象 。

mainloop 告诉窗口去等待用户进行某些操作，虽然现在除了关闭窗口之外，并没有提供给用户多少有用的操作。

创建好一个乌龟（Turle）之后，就可以调用它的一个方法（method）来在窗口中移动。方法和函数类似，但是使用的语法略有不同。例如，要让乌龟向前移动：

```
bob.fd(100)
```

这个方法fd 和我们称为bob 的乌龟对象是关联的。调用方法和发出一个请求类似：你是在请求bob 去向前移动。

fd 的参数是移动的距离，以像素（pixel）为单位，所以实际移动的距离依赖于显示器的分辨率。

Turtle对象的其他方法包括bk（ 用于前进和后退）、lt 和rt（ 用于左转和右转）。lt 和rt 的参数是旋转的角度，单位是度。

另外，每只乌龟都拿着一只笔，可以朝上或者朝下；若笔朝下，则会绘制出走过的路迹。方法pu 和pd 分别表示“笔朝上”（pen up）和“笔朝下”（pen down）。

若要画一个朝右的角，在程序中（建立bob 实例之后，调用mainloop 之前）添加如下代码：

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

运行这个程序时，将会看到bob 先向东走，再向北走，身后留下两条线段。

现在试着修改程序，画出一个正方形来。在成功之前请不要继续！

4.2 简单重复

你可能会写下如下代码：

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

使用for 语句，可以更紧凑地实现同样功能。把下面的例子加到mypolygon.py 中，并再运行一次：

```
for i in range(4):
    print('Hello!')
```

可能会看到如下输出：

```
Hello!
Hello!
Hello!
Hello!
```

这是for 语句的最简单用法，后面我们会看到更多的用法。但这样已经足够重写刚才的画正方形的程序了。请重写后再接着阅读。

下面是使用for 语句绘制正方形的程序：

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

for 语句的语法和函数定义类似。它也有一个以冒号结束的语句头，并有一个缩进的语句体。语句体可以包含任意数量的语句。

for 语句也称为循环 （loop），因为执行流程会遍历语句体，之后从语句体的最开头重新循环执行。在这个例子里，语句体执行了4次。

这个版本的代码和之前的绘制正方形的代码其实还稍有不同，因为在最后一次循环后它多做了一次左转。多余的左转稍微多消耗了点时间，但因为每次循环做的事情都一样，也让代码更简练。这个版本的代码还有一个效果，程序执行完之后，乌龟会回归到初始的位置，并朝向初始相同的方向。

4.3　练习

下面是一系列使用乌龟世界的练习。它们力求有趣，但也包含着某

些寓意。做这些练习时，可以猜想一下其寓意。

在接下来的章节中有这些练习的解答，所以在完成（或着至少尝试过）之前，请先别继续阅读。

1. 写一个函数`square`，接受一个形参`t`，用来表示一只乌龟。利用乌龟来画一个正方形。

写一个函数调用传入`bob`作为实参来调用`square`函数，并再运行一遍程序。

2. 给`square`函数再添加一个形参`length`。修改函数内容，保证正方形的长度是`length`，并修改函数调用以提供这第二个实参。再运行一遍程序。使用不同的`length`值测试你的程序。

3. 复制`square`函数，并命名为`polygon`。再添加一个形参`n`并修改函数体以绘制一个正*n*边形。提示：正*n*边形的拐角是360/*n*度。

4. 写一个函数`circle`接受代表乌龟的形参`t`，以及表示半径的形参`r`，并使用合适的长度和边数调用`polygon`画一个近似的圆。使用不同的`r`值来测试你的函数。

提示：思考圆的周长（circumference），并保证`length * n = circumference`。

另一个提示：如果你觉得`bob`太慢，可以修改`bob.delay`来加速。`bob.delay`代表每次行动之间的停顿，单位是秒。`bob.delay = 0.01`应该能让它跑得足够快。

5. 给circle 函数写一个更通用的版本，称为arc 。增加一个形参angle ，用来表示画的圆弧的大小。这里angle 的单位是度数，所以当arc=360 时，则会画一个整圆。

4.4　封装

第一个练习要求把画正方形的代码放到一个函数定义中，并将乌龟bob 作为实参传入，调用该函数。下面是一个解答：

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

最内侧的语句，fd 和lt 都缩进了两层，表示它们是在for 语句的语句体内部，而for 语句在函数定义的函数体内部。最后一行，square(bob) ，又重新从左侧开始而没有缩进，这表明for 语句和square 函数的定义都已经结束。

在函数体中，t 引用的乌龟和bob 引用的相同，所以t.lt(90) 和直接调用bob.lt(90) 是一样的效果。在这种情况下为什么不直接把形参写为bob 呢？原因是t 可以是任何乌龟，而不仅仅是bob ，所以可以再新建一只乌龟，并将它作为参数传入到square 函数：

```
alice = Turtle()
```

```
square(alice)
```

把一段代码用函数包裹起来，称为封装（encapsulation）。封装的一个好处是，它给这段代码一个有意义的名称，增加了可读性。另一个好处是，当重复使用这段代码时，调用一次函数比复制粘贴代码要简易得多！

4.5 泛化

下一步是给`square`函数添加一个`length`参数。这里是一个解决方案：

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)

square(bob, 100)
```

给函数添加参数的过程称为泛化（generalization），因为它会让函数变得更通用：在之前的版本中，正方形总是一个大小，而新的版本中，可以是任意大小。

下一步也是一次泛化。我们不再只绘制正方形，而是可以绘制任意边数的多边形。这里是一个方案：

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

这个例子绘制一个7边形，边长是70。

如果使用的是Python 2，那么`angle`的值可能会因为整数除法而错误。一个简单的解决办法是使用`angle = 360.0 / n`。因为分子是一个浮点数，所以结果也会是浮点数。

如果函数的形参比较多，很容易忘掉每一个具体是什么，或者忘掉它们的顺序。所以在Python中，调用函数时可以加上形参名称，这样是合法的，并且有时候会有帮助：

```
polygon(bob, n=7, length=70)
```

这些参数被称为关键词参数（keyword argument），因为它们使用“关键词”的形式带上了形参的名称调用（请别和`while`与`def`之类的Python关键字混淆）。

这个语法使得程序更加可读。它也同样提示了我们实参和形参的工作方式：当调用函数时，实参传入并赋值给形参。

4.6 接口设计

下一步是写画圆的circle 函数，接受形参r ，表示圆的半径。下面是一个简单的例子，通过调用polygon 函数画50边的多边形：

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

第一行计算半径为r 的圆的周长，使用公式$2\pi r$ 。因为我们使用的是math.pi ，所以需要先导入math 模块。依照惯例，import 语句一般都放在脚本开头。

n 是我们用于近似画圆的多边形的边数，所以length 是每个边的长度。因此，polygon 画出一个50边形，近似于一个半径为r 的圆。

这个解决方案的缺点之一是n 是一个常量，因此对于很大的圆，多边形的边线太长，而对于小圆，我们又浪费时间去画过短的边线。解决办法之一是泛化这个函数，加上形参n 。这样可以给用户（调用circle 函数的人）更多的控制选择，但接口就不那么清晰整洁了。

函数的接口 是如何使用它的概要说明：它有哪些参数？这个函数做什么？它的返回值是什么？我们说一个接口“整洁”（clean），是说它能够让调用者完成所想的事情，而不需要处理多余的细节。

在这个例子里，r 属于函数的接口，因为它指定了所画的圆的基本属性。相对地，n 则不那么适合，因为它说明的是如何 画圆的细节信息。

所以与其弄乱接口，不如在代码内部根据周长来选择合适的n 值：

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

现在多边形的边数是一个接近circumference/3 的整数，所以每个边长近似是3，已经小到足够画出好看的圆形，但又足够大到不影响画线效率，并且可接受任何尺寸的圆。

4.7 重构

当我写circle 函数时，我可以复用polygon ，因为边数很多的正多边形是圆的很好的近似。但是arc 则并不那么容易对付；我们不能使用polygon 或者circle 来画圆弧。

换个办法，可以先复制一个polygon 函数，再通过修改得到arc 函数。结果可能类似下面的示例：

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
```

```
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length）
        t.lt(step_angle)
```

这个函数的第二部分很像polygon 的实现，但如果不修改polygon 的接口，无法直接复用。我们也可以泛化polygon 函数以接受第三个参数表示圆弧的角度，但那样的话polygon （多边形）就不是合适的名称了！ 所以，我们将这个更泛化的函数称为polyline （多边线）：

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

现在我们可以重写polygon 和arc ，让它们调用polyline ：

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

最后，我们可以重写circle，改为调用arc：

```
def circle(t, r):
    arc(t, r, 360)
```

这个过程——重新组织程序，以改善接口，提高代码复用——被称为重构（refactoring）。在这个例子里，我们注意到arc 和polygon 中有类似的代码，因此我们把它们的共同之处“重构出来”抽取到polyline 函数中。

如果我们早早计划，可能会直接先写下polyline，也就避免了重构，但实际上在工程开始时我们往往并没有足够的信息去完美设计所有的接口。开始编码之后，你会更了解面对的问题。有时候，重构正意味着你在编程中掌握了一些新的东西。

4.8 一个开发计划

开发计划（development plan）是写程序的过程。本章的案例分析中，我们使用的过程是“封装和泛化”。这个过程的具体步骤是：

1．最开始写一些小程序，而不需要函数定义。

2．一旦程序成功运行，识别出其中一段完整的部分，将它封装到一个函数中，并加以命名。

3．泛化这个函数，添加合适的形参。

4．重复步骤1到步骤3，直到得到一组可行的函数。复制粘贴代码，以避免重复输入（以及重复调试）。

5．寻找可以使用重构来改善程序的机会。例如，如果发现程序中几处地方有相似的代码，可以考虑将它们抽取出来做一个合适的通用函数。

这个过程也有一些缺点——我们会在后面看到其他方式——但如果在开始编程时不清楚如何将程序分成适合的函数，这样做会带来帮助。这个方法能让你一边开发一边设计。

4.9　文档字符串

文档字符串（docstring）是在函数开头用来解释其接口的字符串（doc是“文档”documentation的缩写）。下面是一个示例：

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them. t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

依照惯例，所有的文档字符串都使用三引号括起来。三引号字符串又称为多行字符串，因为三引号允许字符串跨行表示。

文档字符串很简洁，但已经包含了其他人需要知道的关于函数的基

本信息。它简明地解释了函数是做什么的（而不涉及如何实现的细节）。它解释了每个形参对函数行为的影响效果以及每个形参应有的类型（如果其类型并不显而易见）。

编写这类文档是接口设计的重要部分。一个设计良好的接口，也应当很简单就能解释清楚；如果你发现解释一个函数很困难，很可能表示该接口有改进的空间。

4.10 调试

函数的接口，作用就像是函数和调用者之间签订的一个合同。调用者同意提供某些参数，而函数则同意使用这些参数做某种工作。

例如，`polyline` 需要4个参数：`t` 必须是一个Turtle；`n` 必须是整数；`length` 应当是个正数；而`angle` 则必须是一个数字，并且按照度数来理解。

这些需求被称为前置条件 ，因为它们应当在函数开始执行之前就保证为真。相对地，函数结束的时候需要满足的条件称为后置条件 。后置条件包含了函数预期的效果（如画出线段）以及任何副作用（如移动乌龟或者引起其他改变）。

满足前置条件是调用者的职责。如果调用者违反了一个（文档说明清晰的！）前置条件，因而导致函数没有正确运行，则bug是在调用者，而不在函数本身。

如果前置条件已经满足，但后置条件没有满足，那么bug就出现在

函数本身。如果前置和后置前置都定义清晰，可以帮助调试。

4.11　术语表

方法（method）：与某个对象相关联的一个函数，使用句点表达式调用。

循环（loop）：程序中的一个片段，可以重复运行。

封装（encapsulation）：将一组语句转换为函数定义的过程。

泛化（generalization）：将一些不必要的具体值（如一个数字）替换为合适的通用参数或变量的过程。

关键词参数（keyword argument）：调用函数时，附带了参数名称（作为一个“关键词”来使用）的参数。

接口（interface）：描述函数如何使用的说明。包括函数的名称，以及形参与返回值的说明。

重构（refactoring）：修改代码并改善函数的接口以及代码质量的过程。

开发计划（development plan）：写程序的过程。

文档字符串（docstring）：在函数定义开始处出现的用于说明函数接口的字符串。

前置条件（precondition）：在函数调用开始前应当满足的条件。

后置条件（postcondition）：在函数调用结束后应当满足的条件。

4.12　练习

练习**4-1**

在http://thinkpython2.com/code/polygon.py下载本章的代码。

1．画一个栈图来显示函数`circle(bob，radius)`运行时的程序状态。你可以手动计算，或者在代码中添加一些`print`语句。

2．在4.7节中的`arc`函数并不准确，因为使用多边形模拟近似圆，总是会在真实的圆之外。因此，Turtle画完线之后会停在偏离正确的目标几个像素的地方。我的解决方案里展示了一种方法可以减少这种错误的效果。阅读代码并考虑是否合理。如果你自己画图，可能会发现它是如何生效的。

练习**4-2**

写一组合适的通用函数，用来画出图4-1所示的花朵图案。

解答：http://thinkpython2.com/code/flower.py，另外也需要http://thinkpython2.com/code/ polygon.py。

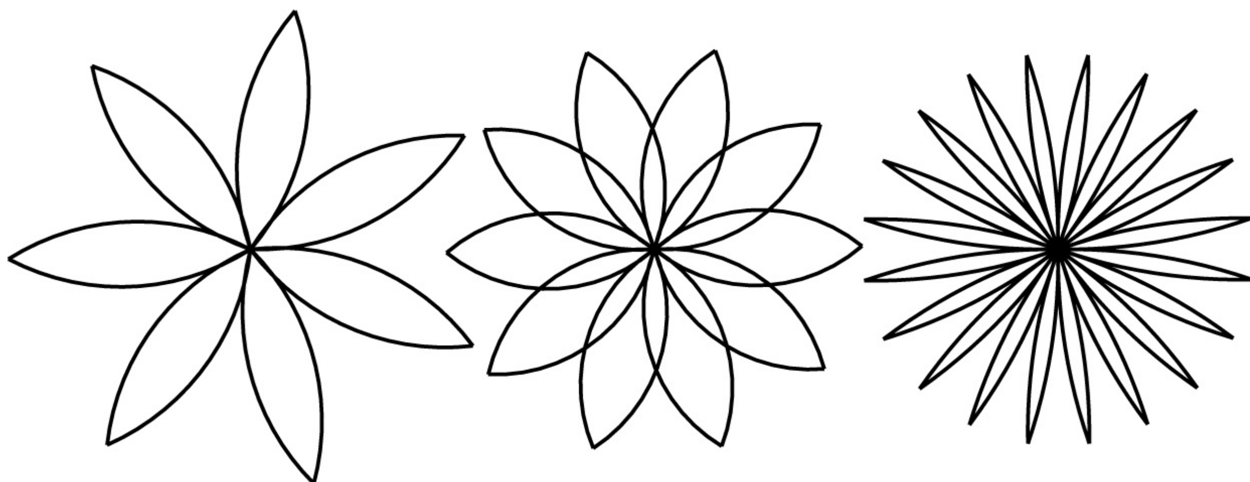


图4-1　花朵图案

练习**4-3**

写一组合适的通用函数，用来画出图4-2所示的图形。

解答：http://thinkpython2.com/code/pie.py。

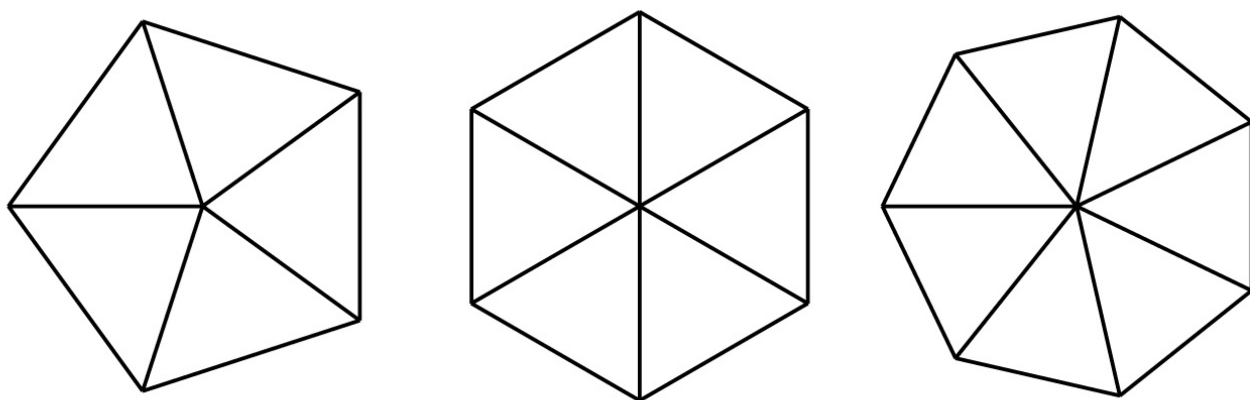


图4-2　饼图

练习**4-4**

字母表中的字母可以使用一些基本元素来构成，如横线、竖线以及一些曲线。设计一个字母表，可以使用最少的基本元素画出来，并编写

函数来画出字母。

你应当给每个字母单独写一个函数，名称为`draw_a`、`draw_b`等，并把这些函数放到`letters.py`文件中。可以从http://thinkpython2.com/code/typewriter.py下载一个“乌龟打字机”程序来帮助测试你的代码。

你可以在http://thinkpython2.com/code/letters.py获得解答，另外也需要http://thinkpython2.com/ code/polygon.py。

练习**4-5**

在http://en.wikipedia.org/wiki/Spiral阅读关于螺旋线（spiral）的信息；接着编写一段程序来画出阿基米德螺旋（或者其他的某种螺旋线）。

解答：http://thinkpython2.com/code/spiral.py。

第5章　条件和递归

本章的主要话题是`if`表达式，它根据程序的状态执行不同的代码。但首先我想要介绍两个新操作符：向下取整除法操作符和求模操作符。

5.1　向下取整除法操作符和求模操作符

向下取整除法操作符（`//`）对两个数进行除法运算，并向下取整得到一个整数。例如，假设一个电影的播放时长为105分钟，你可能会想知道按小时算这是多长。传统的除法会得到一个浮点数：

```
>>> minutes = 105
>>> minutes / 60
1.75
```

但是，我们在写小时数时通常并不用小数点。向下取整除法，则丢弃小数部分，得到整数的小时数：

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

要求得余数，可以从分钟数中减去1小时：

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

另一种办法是使用求模操作符（% ）将两个数相除，得到余数：

```
>>> remainder = minutes % 60
>>> remainder
45
```

求模操作符其实有很多实际用途。例如，可以用它来检测一个数是不是另一个的倍数——如果`x % y`是0，则`x`可以被`y`整除。

另外，也可以用它来获取一个数后一位或后几位数字。例如，`x % 10`可以得到`x`的个位数（10进制）。类似地，`x % 100`可以获得最后两位数。

如果使用的是Python 2，除法机制会有所不同。除法操作符（/）在两个操作数都是整数的情况下，实际进行的是向下取整除法操作，而当两个操作数中有一个是浮点数时，则进行的是浮点数除法。

5.2 布尔表达式

布尔表达式 是值为真或假的表达式。下面的例子中使用了== 操作符，来比较两个操作对象是否相等。如果相等，则得True ， 否则是False：

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True 和False 是类型bool 的两个特殊值；它们不是字符串：

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

== 操作符是一个关系操作符 ；其他的关系操作符有：

```
x != y          # x不等于y
x > y           # x比y大
x < y           # x比y小
x >= y          # x大于或等于y
x <= y          # x小于或等于y
```

虽然你可能对这些操作已经熟悉，但是Python的符号和数学符号还是有些区别的。最常见的错误是使用单等号（= ）而不是双等号（==

）。请记住= 是一个赋值操作符，而== 是一个关系操作符。另外，不存在=< 或者=> 这样的操作符。

5.3 逻辑操作符

逻辑操作符 有3个：and 、or 和not 。这些操作符的语义（意义）和它们在英语中的意思差不多。例如，x > 0 and x < 10 只有当x 比0大且 比10小时才为真。

n%2 == 0 or n%3 ==0，当其中任意 一个条件为真时为真，也就是说，数n 可以被2或3整除都可以。

最后，not 操作符可以否定一个布尔表达式，所以not (x > y) 在x > y 为假时为真，即当x 小于等于y 时真。

严格地说，逻辑操作符的操作对象应该都是布尔表达式，但是Python并不那么严格。任何非0的数都被解释为True 。

```
>>> 42 and True
True
```

这种灵活性可能会很有用，但有时候也会带来一些小困惑。你可能应该避免使用它（除非你很确切地知道自己在做什么）。

5.4 条件执行

为了编写有用的程序，我们几乎总是需要检查条件并据此改变程序的行为的能力。条件语句 给了我们这种能力。最简单的形式是`if` 表达式：

```
if x > 0:
    print('x is positive')
```

`if` 之后的布尔表达式被称为条件 （condition）。如果它为真，则之后缩进的语句会运行。否则，什么都不发生。

`if` 表达式的结构和函数定义一样：一个语句头，接着是缩进的语句体。这种类型的语句称为复合语句 。

语句体中出现的语句数量并没有限制，但是最少需要一行。偶尔可能会遇到需要一个语句体什么都不做（通常是标记一个你还没有来得及写的代码的位置）。这个时候，可以使用`pass` 语句。`pass` 语句什么都不做。

```
if x < 0:
    pass                # TODO: 需要处理负值的情况！
```

5.5 选择执行

`if` 语句的第二种形式是选择执行 ，这种形式下，有两种可能，

而if 的条件决定哪一种运行。语法看起来是这样的：

```
if x%2 == 0:
    print('x is even')
else:
    print('x is odd')
```

如果x 除以2的余数是0，则我们知道x 是偶数（even），并且程序会显示合适的消息even' 。如果条件为假，则第二段语句会运行。因为条件必定是真假之一，所以必然只会有一段语句运行。这两段不同的语句称为分支 （branch），因为它们是程序执行流程中的两个支流。

5.6 条件链

有时候有超过两种的可能，所以我们需要更多的分支。表达这种计算的一种方式是条件链 （chained conditional）：

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elif 是“else if ”的缩写。和之前一样，只有一个分支会运行。elif 语句的数量没有限制。如果有一个else 语句，则它必须放在最后。但

也可以没有else 语句。

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

每个条件都按顺序检查。如果第一个是false，则检查下一个，依此类推。如果有一个条件为真，则运行相应的分支，而整个语句结束。即使有多个条件为真，也只有第一个为真的分支会运行。

5.7 嵌套条件

条件判断可以再嵌套条件判断。我们可以修改前一节中的示例，如下：

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

外侧的条件语句包含两个分支。第一个分支包含一行简单的语句。第二个分支则包含了另一个if 语句，它本身也有两个分支。这两个分

支也都是简单语句，虽然它们其实也可以是条件语句。

虽然语句的缩进让结构非常明晰，但嵌套条件语句 会很快随着嵌套层数增多而变得非常难以阅读。应该尽量避免它。

逻辑操作符常常能够用来简化嵌套条件语句。例如，我们可以将下面的语句替换为单独的一个条件：

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

print 语句只有在两个条件语句都通过时才运行，所以我们可以使用and 操作符达到相同的效果：

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

对于这种类型的条件，Python还提供了一个更简洁的语法：

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8 递归

函数调用另外一个函数是合法的；函数调用自己也是合法的。这样做有什么好处可能还不明显，但它其实是程序能做的最神奇的事情之一。例如，考虑下面的函数：

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n - 1)
```

如果n 是0或负数，它会输出单词“Blastoff!”，其他情况下，它会输出n ，并调用一个名为countdown 的函数——它自己——并传入实参n-1 。

我们调用这个函数时会发生什么？

```
>>> countdown(3)
```

countdown 的执行从n=3 开始，因为n 比0大，所以会输出3，并接着调用自己……

countdown 的执行从n=2 开始，因为n 比0大，所以会输出2，并接着调用自己……

countdown 的执行从n=1 开始，因为n 比0大，所以会输出

1，并接着调用自己……

countdown 的执行从n=0 开始，因为n 不比0大，所以会输出单词“Blastoff!”，并返回。

接收n=1 的函数countdown 返回。

接收n=2 的函数countdown 返回。

接收n=3 的函数countdown 返回。

然后就会到了__main__ 函数。所以，全部的输出如下：

```
3
2
1
Blastoff!
```

调用自己的函数称为递归的 （recursive）函数，这个执行的过程叫作递归 （recursion）。

另外举一个例子，我们可以写一个函数打印某个字符串n 次。

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

如果`n <= 0`，`return`语句会直接退出当前函数。执行流程会立即返回到调用者，之后的语句不会运行。

函数另外的部分和`countdown`类似：如果`n`大于0，它会打印`s`并且调用自己，以再进行$n-1$次显示`s`的操作。所以输出的行数是`1+(n-1)`，也就是`n`。

对于这样简单的例子来说，可能使用`for`循环会更容易。但我们会在后面见到一些示例，使用`for`循环很难写，但使用递归则会很简单，所以早早开始了解递归是件好事。

5.9　递归函数的栈图

在3.10节中，我们使用一个栈图来表示程序在进行函数调用时的状态。同样的栈图，可以用来帮助我们解释递归函数。

一个函数每次被调用时，Python会创建一个帧（function frame），来包含函数的局部变量和参数。对于递归函数，栈上可能同时存在多个函数帧。

图5-1展示了`countdown`函数在`n=3`调用时的栈图。

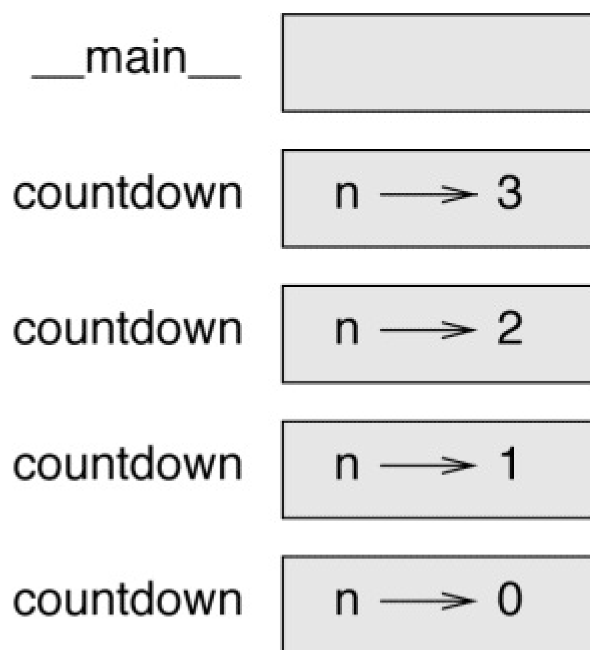


图5-1　栈图

和往常一样，栈的顶端是__main__ 的函数帧。因为我们没有在__main__ 函数里新建任何变量或传入任何参数，所以它是空的。

4个countdown 函数帧有不同的参数n 值。最底端的栈，其n=0 ，被称为基准情形 （base case） 。因为它不再进行递归调用，所以后面没有其他函数帧了。

作为练习，为函数print_n 画一个栈图，其调用实参是s = 'Hello' 和n=2 。然后写一个函数do_n ，接受一个函数对象和一个数字n 作为形参。它会调用给定的函数n 次。

5.10　无限递归

如果一个递归永远达不到基准情形，则它会永远继续递归调用，而

程序也永不停止。这个现象被称为无限递归，而它并不是个好主意。下面是一个会引起无限递归的最简单函数：

```
def recurse():
    recurse()
```

在大多数程序环境中，无限递归的函数并不会真的永远执行。Python会在递归深度到达上限时报告一个出错消息：

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

这个调用回溯比上一章看到的要大一些。当这个错误发生时，栈上已经有1000个`recurse` 帧了！

如果你不小心写出了一个无限循环，请复查自己的函数，确认里面至少有一个基准情形不进行递归调用。如果已经有了一个基准情形，检查是否已经确保在运行时能达到它。

5.11 键盘输入

目前为止我们写过的程序都还不能接收用户的输入。它们只能每次做相同的事情。

Python提供了一个内置函数`input` 来从键盘获取输入并等待用户输入一些东西。当用户按下回车键，程序会恢复运行，而且`input` 则通过字符串形式返回用户输入的内容。在Python 2里，这个函数叫`raw_input` 。

```
>>> text = input()
Whate are you waiting for?
>>> text
Whate are you waiting for?
```

在从用户那里获得输入之前，最好打印一个提示信息，告诉用户希望他们输入什么。`raw_input` 函数可以接受一个参数作为提示：

```
>>> name = input('What...is your name?\n')
What...is your name?
Authur, King of the Britons!
>>> name
Authur, King of the Britons!
```

提示信息最后的`\n` 表示一个换行符，它是会引起输出显示换行的特殊字符。这也是为何用户的输入显示在提示信息的下一行的原因。

如果希望用户输入一个整数，可以尝试将输入值转换为`int` ：

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

但如果用户输入不是数字的话，会得到错误：

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

后面我们会看到如何处理这种错误。

5.12　调试

当发生语法错误和运行时错误时，出错消息包含了大量的信息，但有时候反而会信息过量。最有用的信息是：

- 错误的类型；
- 发生错误的地方。

语法错误通常都很容易定位，但也有棘手之处。空格问题引起的错误很难处理，因为空格和制表符都是不可见的，我们已经习惯于忽视它

们。

```
>>> x = 5
>>>  y = 6
   File "<stdin>", line 1
      y = 6
      ^
IndentationError: unexpected indent
```

这个例子中，问题的原因是第二行多缩进了一个空格。但出错消息指向的是y，容易误导。总的来说，出错消息会告诉我们发现错误的地址，但真正发生的地方可能在更前面的代码中，有时候甚至在前一行。

运行时错误也是如此。假设你想要按照分贝来计算信噪比。公式是$SNR_{db} = 10 \lg(P_{signal} / P_{noise})$。在Python中，你可能会这么写：

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

在运行这个程序时，会得到一个异常：

```
Traceback (most recent call last):
   File "snr.py", line 5, in ?
       decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

出错消息指向第5行，但那一行其实没有什么错误。要找到真正的错误，可能需要打印出`ratio`的值，结果你会发现是0。问题出在第4行，这里使用了向下取整除法而不是浮点数除法。

你应该花一些时间认真阅读出错消息，但不要认为出错消息上说的每一样都对。

5.13　术语表

向下取整除法（floor division）：用`//`表示的操作符，用于将两个数相除，并对结果进行向下取整（靠近0取整），得到整数结果。

求模操作符（modulus operator）：用`%`表示的操作符，用于两个整数，返回两数相除的余数。

布尔表达式（boolean expression）：一种表达式，其值是`True`或`False`。

关系操作符（relational operator）：用来表示两个操作对象的比较关系的操作符，如下之一：`==`、`!=`、`>`、`<`、`>=`和`<=`。

逻辑操作符（logical operator）：用来组合两个布尔表达式的操作符，有3个：`and`、`or`和`not`。

条件语句（conditional statement）：依照某些条件控制程序执行流程的语句。

条件 （condition）：条件语句中的布尔表达式，由它决定执行哪一个分支。

复合语句 （compound statement）：一个包含语句头和语句体的语句。语句头以冒号（:）结尾。语句体相对语句头缩进一层。

分支 （branch）：条件语句中的一个可能性分支语句段。

条件链语句 （chained conditional）：一种包含多个分支的条件语句。

嵌套条件语句 （nested conditional）：在其他条件语句的分支中出现的条件语句。

返回语句 （return statement）：导致一个函数立即结束并返回到调用者的语句。

递归 （recursion）：在当前函数中调用自己的过程。

基准情形 （base case）：递归函数中的一个条件分支，里面不会再继续递归调用。

无限递归 （infinite recursion）：没有基准情形的递归，或者永远无法达到基准情形的分支的递归调用。最终，这种无限递归会导致运行时错误。

5.14　练习

练习**5-1**

`time` 模块提供了一个函数，名字也叫`time` ，它能返回从“纪元”起到当前的格林尼治时间。“纪元”其实是人为选作基准点的时间。在UNIX系统中，纪元时间点是1970年1月1日。

```
>>> import time
>>> time.time()
1437746094.5735958
```

编写一个脚本，读取当前时间，并转换为一天中的小时数、分钟数、秒数，以及从纪元起到现在的天数。

练习**5-2**

费马大定理是说对于任何大于2的n ，不存在任何正整数a 、b 和c 能够满足：

$$a^n + b^n = c^n$$

1．编写一个函数`check_fermat` ，接收4个形参（即`a` 、`b` 、`c` 和`n` ）并检查费马定理是否成立。如果n 比2大并且满足

$$a^n + b^n = c^n$$

则程序应当打印“天哪，费马弄错了！”，否则程序应当打印“不，那样不行”。

2．编写一个函数，提示用户输入a 、b 、c 和n 的值，将它们转换为整数，并使用check_fermat 来检查它们是否违背了费马定理。

练习 **5-3**

如果给你3根木棍，你可能可以将它们摆成一个三角形，也可能不可以。例如，如果一根木棍的长度是12英寸，而其他两根都只有1英寸，那么你无法让短的木棍在中间相接。对于任意3个长度，有一个简单的测试可以检验它们是否可能组成一个三角形：

如果其中有任意一个长度的值大于其他两个长度的和，则你不能组成三角形。否则可以。（如果两个长度的和等于第三个，则它们组成一个“退化”的三角。）

1．编写一个函数is_triangle ，接收3个整数参数，并根据这组长度的木棍是否能组成三角形来打印“Yes”或“No”。

2．编写一个函数提示用户输入3根木棍的长度，将其转换为整数，并使用is_triangle 检查这些长度的木棍是否可以组成三角形。

练习**5-4**

下面的程序的输出是什么？画一个栈图来显示程序打印结果的时候的状态。

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)
```

```
recurse(3, 0)
```

1．如果像`recurse(-1, 0)`这样调用这个函数，会发生什么？

2．编写一段文档字符串，向人解释清楚要使用这个函数需要知道的东西（并且不多写其他内容）。

3．接下来的练习需要使用第4章描述的`turtle`模块。

练习**5-5**

阅读下面的函数，并看看你能否弄清楚它在做什么（参看第4章中的示例）。接着运行它，看你的理解是否正确。

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

练习**5-6**

科赫曲线（Koch curve）是一个分形，它看起来像图 5-2所示。要

绘制一个长度为*x* 的科赫曲线，你只需要做：

1．绘制长度为*x* /3的科赫曲线。

2．向左转60°。

3．绘制长度为*x* /3的科赫曲线。

4．向右转120°。

5．绘制长度为*x* /3的科赫曲线。

6．向左转60°。

7．绘制长度为*x* /3的科赫曲线。

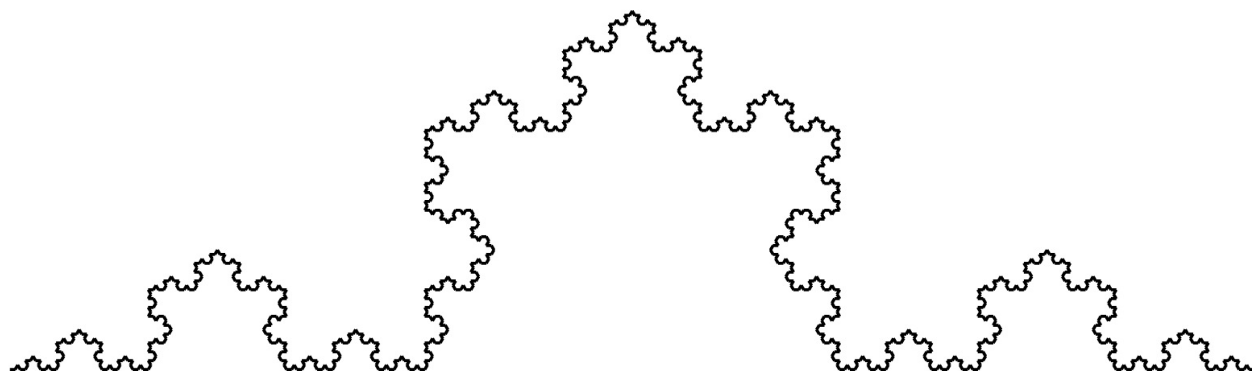


图5-2　一个科赫曲线

当*x* 比3小的时候例外：在那种情况下，你可以直接绘制一个长度为*x* 的直线。

1．编写一个函数koch，接收一个Turtle以及一个长度作为形参，并使用Turtle绘制指定长度的科赫曲线。

2．编写一个函数`snowflake`，绘制3条科赫曲线，组成一个雪花形状。解答：http://thinkpython2.com/code/koch.py。

3．科赫曲线可以用几种方法泛化。参看http://en.wikipedia.org/wiki/Koch_snowflake中的例子，并实现你最喜欢的一个。

第6章　有返回值的函数

我们用过的很多Python函数（如数学函数）都会产生返回值。但是，到目前为止我们写的函数都是没有返回值的：它们只产生一个效果，如打印某些值或者移动乌龟，但是并不返回值。在本章中你将学会如何写有返回值的函数。

6.1　返回值

调用函数会产生一个返回值，我们一般会将它赋值给一个变量或者用作表达式的组成部分：

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

目前为止我们写的函数都是无返回值的函数。用通俗的话说，它们没有返回值；用更精确的话说，它们返回的值是None 。

本章中，我们会（终于）写一些有返回值函数。第一个例子是area，用于计算给定半径的圆的面积：

```
def area(radius):
    a = math.pi * radius**2
    return a
```

之前我们已经见过return 语句，但在有返回值函数中，return 语句包含了一个表达式。这个语句的意思是：“立即从这个函数中返回，并使用后面的表达式作为返回值。”表达式可以任意复杂，所以我们可以把这个函数写得更紧凑：

```
def area(radius):
    return math.pi * radius**2
```

另一方面，类似于a 这样的临时变量 常常会让调试更容易。

有时候函数中针对不同的条件分支，各有各的返回语句会很有用处：

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

因为return 语句分别在不同的分支中，只有一个运行。

一旦return 语句运行，当前的函数就会终结，后面的语句不会执行。return 语句之后的代码，或者在其他程序流程永远不可能达到的地方的代码，称为无效代码 （dead code）。

在有返回函数中，保证每个可能执行路径上都会遇到return 语句，是个很好的主意。例如：

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

这个函数并不正确，因为如果x 正好是0，则两个条件都不为true，则此时函数会没有遇到return 语句就终结了。如果执行流程到了函数的结尾，返回值是None ，它并不是0的绝对值。

```
>>> absolute_value(0)
None
```

顺便说一下，Python内置提供了计算绝对值的函数abs 。

作为练习，写一个compare 函数，带两个参数x 和y ，如果x > y ，返回1，如果x == y ，返回0，如果x < y ，返回-1。

6.2　增量开发

当你写更复杂的函数时，可能会发现需要更多的时间来调试。为了对应不断增加的程序复杂度，你可能会想尝试一下称为增量开发 的过

程。增量开发的目标是通过每次只增加和测试一小部分代码，来避免长时间的调试过程。

例如，假设你想要查找两点之间的距离，给定坐标(x_1, y_1)和(x_2, y_2)。根据毕达哥拉斯定理，距离是：

$$距离 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

第一步考虑Python中`distance`函数应该是什么样子的。换句话说，输入（形参）是什么？输出（返回值）是什么？

在这个例子中，输入是两个点，并可以用4个数字来表示。返回值是距离，它用一个浮点数表示。

现在就可以写出函数的轮廓了：

```
def distance(x1, y1, x2, y2):
    return 0.0
```

显然，现在这个版本计算的并不是距离；它总是返回0。但它是语法结构正确的，并且能运行，即意味着你可以在继续开发更复杂的功能之前对它进行初步的测试。

要测试这个新函数，使用样本参数调用它：

```
>>> distance(1, 2, 4, 6)
0.0
```

我选择这些值，因为这样两个点之间，横向距离是3，纵向距离是4；也就是，结果是5（3-4-5直角三角形的斜边）。当测试一个函数时，事先知道正确的结果是很有用的。

到这个时候我们已经确认函数的语法形式是正确的，紧接着可以给函数体添加代码了。合理的下一个步骤是找到距离差$x_2 - x_1$和$y_2 - y_1$。下一版本的函数将这两个距离差保存到临时变量中并打印出来。

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

如果函数正确执行，应该会显示‘dx is 3’和‘dy is 4’。如果确实如此，我们就可以确认函数正确地获得了实参，并正确地执行了第一步计算。如果不是如此，则只有几行代码需要检查。

下一步我们计算dx 和dy 的平方和：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

同样地，你可以在这里再运行一遍程序，并检查输出（应该是25）。最后，可以使用`math.sqrt`来计算并返回结果：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

如果这个函数运行正确，那么你的任务就完成了。否则，你可能需要在`return`语句之前打印出`result`的值。

最终版本的函数运行时并不打印任何东西；它只会返回一个值。我们之前写的`print`语句在调试时很有用，但一旦你的函数编写正确，就应该删除掉它们。这种代码称为脚手架代码 （scaffolding），因为它们在构建程序的过程中很有用，但并不是最终产品的一部分。

开始的时候，应当每次只添加一到两行代码。当你获得更多经验之后，就会发现自己可以编写和调试更多的代码了。不论如何，增量开发都能帮你节省大量的调试时间。

这个过程有以下几个关键点。

1．以一个可以正确运行的程序开始，每次只做小的增量修改。如果在任意时刻发现错误，你都应当知道错在哪里。

2．使用临时变量保存计算的中间结果，你可以显示和检查它们。

3．一旦整个程序完成，你可能会想要删除掉某些脚手架代码或者把多个语句综合到一个复杂表达式中。但只在不会增加代码阅读的难度时才应该那么做。

作为练习，使用增量开发来编写一个函数hypotenuse ，给定直角三角形的另外两个直角边的长度时，它返回斜边的长度。开发过程中，记录每一步的情况。

6.3 组合

你可能已经想到，在一个函数中可以调用另外一个函数。作为示例，我们会写一个函数，它接收两个点，圆心和圆周上的一点，并计算圆的面积。

假设圆心保存在变量xc 和yc 中，而圆周上的点保存在xp 和yp 上。第一步是算出圆的半径，也就是这两个点的距离。我们刚才已经写了一个函数，distance ，正好有这个功能：

```
radius = distance(xc, yc, xp, yp)
```

第二步是使用上一步算出来的半径来计算圆的面积。我们刚才也写了这个函数：

```
result = area(radius)
```

将这两步封装成一个函数，我们得到：

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

临时变量radius 和result 在开发和调试时有用，可一旦程序已经可以正确运行，我们就可以使用函数组合来简化函数：

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 布尔函数

函数可以返回布尔值，这样可以很方便地隐藏函数内复杂的检测。例如：

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

通常布尔函数的命名都类似于是/否的问句。is_divisible 返回 True 或False ，表示x 是否可以被y 整除。

这里是一个例子：

```
>>>  is_divisible(6, 4)
False
>>>  is_divisible(6, 3)
True
```

== 操作符的结果是一个布尔值，所以我们可以把这个函数写得更加紧凑：

```
def is_divisible(x, y):
    return x % y == 0
```

布尔函数常常用在条件语句中：

```
if is_divisible(x, y):
    print('x is divisible by y')
```

你可能想这么写：

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

但这里多出来的比较是不必要的。

作为练习，写一个函数`is_between(x, y, z)`，当$x \le y \le z$时，返回`True`，其他情况返回`False`。

6.5 再谈递归

至今为止，我们只涉及Python的一个很小的子集，但你可能会有兴趣知道，这个子集已经是一个完备 的编程语言，也就是说，任何可以计算的问题，都可以用这个子集语言来完成。任何已有的程序，都可以用现在已经学会的语言特性重写出来（实际上，可能还需要一些命令来控制诸如键盘、鼠标、光盘之类的设备，但仅此而已）。

要证明这个论断，并不是简单的工作。这个证明最早是由第一代计算机科学家之一阿兰·图灵（Alan Turing）完成的（有人会争辩他其实是个数学家，但大部分早期的计算机科学家都是从数学家开始的）。因此，这被称为图灵论题（Turing Thesis）。若想了解关于图灵论题的更完整（更准确）的讨论，我推荐Michael Sipser的《计算理论导引》（*Introduction to the Theory of Computation*，Course Technology，2012）一书。

为了初步了解如何使用我们现在学会的工具，可以考虑几个递归定

义的数学函数。递归定义和循环定义有些相似，因为同样地，定义中都会包含要定义的事物本身。真正的循环定义往往没什么用：

vorpal：

一个形容词，用来描述一个vorpal的事物。

如果你在词典中看到这样的定义，可能会感觉恼怒。另一方面，如果你查看阶乘函数（用!表示）的定义，可能会看到如下内容：

$0! = 1$

$n! = n(n - 1)!$

这个定义说明0的阶乘是1，而任意其他值n的阶乘是$n - 1$的阶乘乘以n。

所以3!是3乘以2!，而2!是2乘以1!，而1!是1乘以0!。综合起来，3!等于3乘以2乘以1乘以1，即6。

如果能够使用递归定义来描述一个事物，那么也可以使用Python程序来计算它。第一步是决定使用什么形参。在这个例子里，很明显函数`factorial`需要一个整数形参：

```
def factorial(n):
```

如果实参正好是0，我们只需要直接返回1：

```
def factorial(n):
    if n == 0:
        return 1
```

否则，接下来是有意思的地方，我们需要递归调用函数来计算$n-1$的阶乘，并乘以n：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

这个程序的运行流程和5.8节里的countdown函数类似。如果我们使用实参值3调用factorial：

因为3不是0，我们使用第二个分支，计算n-1的阶乘……

因为2不是0，我们使用第二个分支，计算n-1的阶乘……

因为1不是0，我们使用第二个分支，计算n-1的阶乘……

因为0等于0，我们使用第一个分支并返回1，不再需要进行任何递归调用了。

返回值（1）乘以n=1，结果返回。

返回值（1）乘以n=2，结果返回。

返回值（2）乘以n=3，结果是6，而这个结果就是整个函数的返回值。

图6-1显示了这一系列函数调用的栈图。

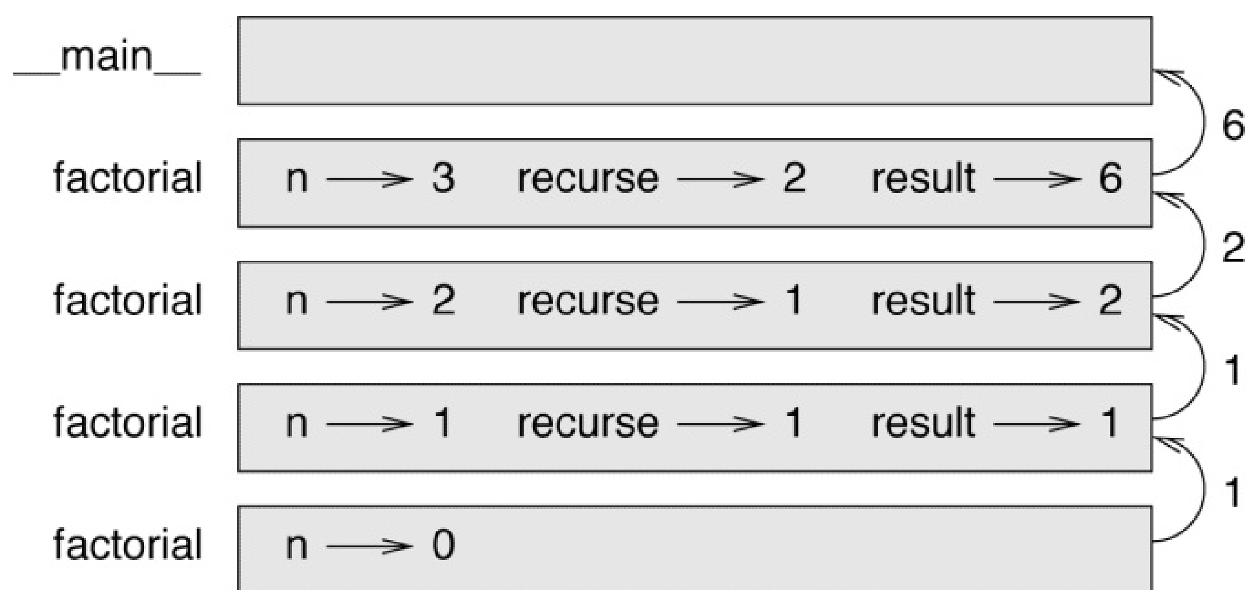


图6-1　栈图

结果值在图中显示为沿着栈向上回传。在每个帧中，返回值是result 的值，即n 和recurse 的乘积。

最后一帧中，局部变量recurse 和result 不存在，因为新建它们的分支并没有运行。

6.6　坚持信念

跟踪程序执行的流程是阅读程序的一个办法，但那样很快就会陷入

迷宫境况。另外有个办法，我称为“坚持信念”。在遇到一个函数调用时，不去跟踪执行的流程，而假定 函数是正确工作的，能够返回正确的结果。

事实上，在使用内置函数时，你已经在这样尝试着坚持信念了。当调用用`math.cos` 或`math.exp` 时，你并不去检查那些函数的内部实现。你只会假定它们是正确的，因为写这些内置函数的一定是很优秀的程序员。

当调用自己写的函数时，这个道理也成立。例如，在6.4节中，我们写了`is_divisible` 函数用来判断一个数是否可以被另一个数整除。一旦我们说服自己认定这个函数是正确的——通过检查代码和测试——就可以直接使用它，而不需要再细看内部实现了。

对递归函数来说，也是如此。当调用递归函数时，不需要检查执行的流程，你应该假定递归调用是正确的（返回正确的结果），并问自己：“假设我能够正确得到$n-1$的阶乘，如何计算n 的阶乘？”很明显你可以做到，直接乘以n 即可。

当然，在还没有完成函数的编写时，就假设它能正确工作，看起来有些奇怪，但那也正是为什么我称它为“坚持信念”的原因！

6.7　另一个示例

除阶乘`factorial` 之外，最常见的递归数学定义是斐波那契数列（`fibonacci` ），其定义如下（参见 http://en.wikipedia.org/wiki/Fibonacci_number）：

fibonacci(0) = 0

fibonacci (1) = 1

fibonacci (*n*) = fibonacci (*n* - 1) + fibonacci (*n* - 2)

翻译成Python后，看起来是这样：

```
def fibonacci (n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

如果你在这里试图跟踪执行的流程，即使是很小的参数*n* ，都会感觉头都要炸了。但因为坚持信念，如果你假定两个递归调用都正常工作，那么很明显，把它们加到一起必然得到正确的结果。

6.8 检查类型

如果我们调用`factorial` 函数，并给定1.5作为实参，会发生什么呢？

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

看起来像是无限递归。怎么会这样？函数中有一个基准情形——当`n == 0`时。但如果`n`不是整数，我们就可能错过 这个基准情形，而永远递归下去。

在第一个递归调用中，`n`是0.5。第二个，`n`是−0.5。从此以后，它会越来越小（更小的负数），但永远不会变成0。

我们有两个选择。可以尝试泛化函数`factorial`，使之能正确处理浮点数，或者我们也可以让`factorial`检查其实参的类型。第一个选择在数学上叫作伽玛函数（gamma function），它有些超出了本书的范围。所以我们选择第二个。

我们可以使用内置函数`isinstance`来检查实参的类型。与此同时，我们也可以确保实参是正数：

```
def factorial (n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

第一个基准情形处理非整数，第二个处理负数。这两种情况中，程序打印一个错误信息，并返回`None`，表示出现了问题：

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

如果我们通过了这两个测试，就能确保知道*n* 是正数或0，所以我们可以证明递归必然终结。

这个程序演示了一个模式，它有时被称为守卫（guardian）。前两个条件就像守卫一样，保护后面的代码，以免出现错误。守卫使得证明代码的正确性成为可能。

在11.3节中我们会看到一个更灵活的方案，用以打印错误信息：抛出一个异常。

6.9　调试

将一个大程序分解为小函数，自然而然地引入了调试的检查点。如果一个函数不能正常工作，可以考虑3种可能性。

- 函数获得的实参有问题，某个前置条件没有达到。
- 函数本身有问题，某个后置条件没有达到。
- 函数的返回值有问题，或者使用的方式不正确。

要排除第一种可能，可以在函数开始的地方加上`print` 语句，显示

实参的值（以及它们的类型）。或者可以添加代码来显式地检查前置条件。

如果实参看起来没错，在每个return 语句前添加print 语句，显示返回值。如果有可能，手动检查返回值。考虑使用能更容易检验结果的实参来调用函数，就像6.2节中的那样。

如果函数看起来正常，检查调用它的代码，确保返回值被正确使用（或者确实被使用了！）。

在函数的开端和结尾处增加print 语句，能帮助我们更清晰地了解函数的执行流程。例如，这里是一个添加了print 语句的factorial 函数：

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

space 是一个字符串，包含多个空格，用来控制输出内容的缩进。下面是调用factorial(4) 的结果：

```
                 factorial 4
             factorial 3
```

```
            factorial 2
        factorial 1
factorial 0
returning 1
        returning 1
            returning 2
                returning 6
                    returning 24
```

如果你对函数调用的流程有困惑，这种输出可以帮你。开发有效的脚手架代码需要花费时间，但一点点脚手架可以节省大量的调试。

6.10　术语表

临时变量（temporary variable）：在复杂计算中用于保存中间计算值的变量。

无效代码（dead code）：程序中的一些代码，永远不可能运行。常常是写在`return`语句之后的代码。

增量开发（incremental development）：一个程序开发计划，通过每次只增加少量代码，并加以测试的步骤，来减少调试。

脚手架代码（scaffolding）：在开发过程中使用的，但在最终版本中不需要的代码。

守卫（guardian）：一个编程模式。使用条件语句来检查并处理可能产生错误的情形。

6.11 练习

练习6-1

为下面的程序绘制一个栈图。程序的输出是什么？

```
def b(z):
     prod = a(z, z)
     print(z, prod)
     return prod

def a(x, y):
     x = x + 1
     return x * y

def c(x, y, z):
     total = x + y + z
     square = b(total)**2
     return square

x = 1
y = x + 1
print c(x, y+3, x+y)
```

练习 6-2

Ackermann函数，$A(m, n)$，定义如下：

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1, A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

参考http://en.wikipedia.org/wiki/Ackermann_function。编写一个函

数ack，计算Ackermann函数的值。使用你的函数求ack(3, 4) 的值，它应当是125。对于很大的数字m 和n，会发生什么？

解答：http://thinkpython2.com/code/ackermann.py。

练习 **6-3**

回文是一个正向和逆向拼写都相同的单词，例如“noon”和“redivider”。递归地说，如果一个单词第一个和最后一个字母相同，并且中间是一个回文，则该单词是回文。

下面的函数接收一个字符串形参并返回第一个、最后一个以及中间的字母：

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

在第8章中查看它们是如何使用的。

1. 将这些函数保存到一个文件palindrome.py 中并测试它们。如果你使用一个包含两个字母的字符串调用middle，会发生什么？使用一个字母呢？空字符串呢？空字符串写作'　'并且不包含任意字母。

2．编写一个函数`is_palindrome`，接收一个字符串形参，并当它是回文的时候返回`True`，否则返回`False`。记着你可以使用内置函数`len`来检测字符串的长度。

解答：http://thinkpython2.com/code/palindrome_soln.py。

练习 **6-4**

我们说一个数*a*是*b*的乘方，如果*a*可以被*b*整除，并且*a*/*b*也是*b*的乘方。编写一个函数`is_power`接收形参`a`和`b`，当`a`是`b`的乘方时返回`True`。注意：你需要考虑基准情形。

练习 **6-5**

*a*和*b*的最大公约数（GCD）是它们两个都能整除的最大的数。

寻找两个数的最大公约数的方法之一是基于如下观察：如果*r*是*a*除以*b*的余数，则gcd(*a*, *b*) = gcd(*b*, *r*)。作为基准情形，我们可以使用gcd(*a*, 0) = *a*。

编写一个函数`gcd`，接收形参`a`和`b`，并返回它们的最大公约数。

鸣谢：这个练习是基于Abelson和Sussman的《计算机程序的构造和解释》（*Structure and Interpretation of Computer Programs*）（MIT出版社，1996）一书。

第7章　迭代

本章讲关于迭代的话题。迭代即重复运行一段代码语句块的能力。我们在5.8节见过一种使用递归来进行的迭代，在4.2节中见过另一种使用`for` 循环进行的迭代。在本章中我们将会看到使用`while` 循环进行的第三种迭代。首先我们先进一步讲讲变量赋值的话题。

7.1　重新赋值

你应当已经发现，对一个变量进行多次赋值是合法的。新的赋值语句使现有的变量引用一个新值（并不再引用老值）。

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

因为第一次显示x 时，它的值是5，而第二次时它的值是7。

图7-1显示了在状态图中，重新赋值 是什么样子的。

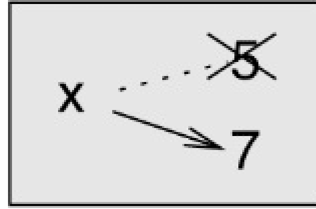


图7-1　状态图

在这里我想解释一个常见的误区。因为Python使用等号（= ）来表示赋值，故很容易将a = b 这样的赋值语句错误理解为数学中表示a 和b 相等的命题。这样理解是错误的。

首先，相等判断是个对称的关系，而赋值并不是。例如，在数学中，如果*a* =7那么7=*a* 。但是在Python中，语句a = 7 是合法的，但7 = a 则是非法的。

另外，在数学中，一个相等判断的命题总是非真即假。如果现在*a* =*b* ，那么*a* 总会等于*b* 。在Python中，赋值语句会让两个变量变得相等，但它们并不会总保持那个状态：

```
>>> a = 5
>>> b = a         # a和b现在相等了
>>> a = 3         # a和b不再相等了
>>> b
5
```

第三行修改a 的值，但是并不会修改b 的值，所以它们不再相等。

虽然重新赋值常常很有用处，但是应该谨慎使用。如果变量的值经常变化，会导致程序难以阅读和调试。

7.2　更新变量

重新赋值的最常见形式是更新，此时变量的新值依赖于旧值。

```
>>> x = x + 1
```

这个语句的意思是“获取x 的当前值，加一，再更新x 为此新值”。

如果尝试更新一个并不存在的变量，则会得到错误，因为Python在赋值给x 之前会先计算等号右边的部分：

```
>>> x = x + 1
NameError: name 'x' is not defined
```

在更新变量之前，必须先对它进行初始化 。通常通过一个简单赋值操作来进行初始化：

```
>>> x = 0
>>> x = x + 1
```

通过加1来更新一个变量，称为增量（increment）；减1的操作称为减量（decrement）。

7.3　while 语句

计算机常被用来自动化重复处理某些任务。重复执行相同或相似的任务，而不犯错误，这是电脑所擅长于人之处。在计算机程序中，重复也被称为迭代 。

我们之前已经看到两个函数countdown 和print_n ，它们使用递归来进行迭代操作。由于迭代如此常见，Python提供了语言特性来支持它。其中一个是我们在4.2节中见过的for 循环语句。后面我们会再回到这个话题。

另一个则是while 语句。下面是使用while 语句实现的countdown 函数：

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Blastoff!')
```

你基本上可以按照英语来理解while 语句。它的意思是：“每当n 还大于0时，显示n 的值，并将n 减1。当n 变成0的时候，显示单词Blastoff! 。”

用更正式的说法，下面是while 语句执行的流程。

1．确定条件是真还是假。

2．如果条件为假，退出while 语句，并继续执行后面的语句。

3．如果条件为真，则运行while 语句的语句体，并返回第1步。

这种类型的流程称为循环 （loop），因为第3步又循环返回到最顶端的第1步了。

循环的语句体里面应当修改一个或多个变量的值，以致循环的条件最终能变成假，而退出循环。否则这个循环会永远重复下去，这样的情况叫作无限循环 。计算机科学家在读到洗发液的说明“涂抹、冲洗、重复”时，总会感到有趣，因为这是一个无限循环。

在countdown 这个例子里，我们可以证明循环必然终结：如果 n 是0或负数，该循环从不运行。否则，n 的值都会减小，因此最终n 会变成0。

对于某些循环，就并不一定那么容易判断了。例如：

```
def sequence(n):
    while n != 1:
          print(n)
          if n % 2 == 0:            # n是偶数
               n = n / 2
          else:                    # n是奇数
               n = n*3 + 1
```

这个循环的条件是n != 1，所以只要n 还没有变成1而导致条件变假，循环就会一直进行下去。

每一个循环中，程序输出n 的值，并检查它是偶数还是奇数。如果是偶数，n 会除以2。如果是奇数，n 会被替换为n*3+1 。例如，如果传入sequence 函数的参数是3，则n 的结果值是：3, 10, 5, 16, 8, 4, 2, 1。

因为n 有时候增加，有时候减少，没有办法找到明显的证据确定n 一定会最终变成1，或者说程序会终止。对于某些特定的n 值，我们可以证明最终会终止。例如，如果开始的参数值是2的幂方，则每次循环n 都是偶数，直到变成1。前面的例子中有一部分就是这样的序列，以16开始。

但困难的问题是，我们是否能够证明这个程序对所有的正值 n 都可以最终终止。至今为止，还没有人对这个问题给出证明或证伪！（参见http://en.wikipedia.org/wiki/Collatz_ conjecture）。

作为练习，重写5.8节中的print_n 函数，使用循环而非递归来实现。

7.4 break 语句

有时候只有在循环语句体的执行途中才能知道是不是到了退出循环的时机。这时候可以使用break 语句来跳出循环。

例如，假设你想要获得用户输入，直到他们输入done 。可以这么写：

```
while True:
    line = input('> ')
    if line == 'done':
        break
```

```
    print(line)

print('Done!')
```

循环的条件是True，总是为真，所以循环会一直运行，直到遇到break 语句。

每次循环之内，都会先用一个尖括号（>）来提示用户输入。如果用户输入done，break 语句会退出循环。否则程序会显示出用户输入的内容，并重新回到循环的顶端。这里是一个运行的实例：

```
> not done
not done
> done
Done!
```

这种写while 循环的方式很常见，因为可以把判断循环条件的逻辑放在循环中的任何地方（而不只是在顶端），并且可以以肯定的语气来表示终结条件（“当这样发生时停止循环”），而不是否定的语气（“继续执行，直到那个条件发生”）。

7.5　平方根

程序中常常使用循环来进行数值计算，以一个近似值开始，并迭代地优化计算结果。

例如，计算平方根的方法之一是牛顿方法。假设你想要知道a 的平方根。如果你以任意一个估计值x 开始，可以使用如下的方程获得一个更好的估计值。

$$y = \frac{x + a/x}{2}$$

例如，如果a 是4而x 是3：

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

这个结果更接近正确的答案（$\sqrt{4} = 2$）。如果我们使用新的估计值重复这个过程，会得到更近似的结果：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

再经过几次重复更新，估计值会几乎完全准确：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
```

```
>>> y
2.00000000003
```

通常来说，我们并不能提前知道需要多少步才能得到正确的答案，但是当估计值不再变化时，我们就知道达到目的了。

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

当`y == x`时，可以终止。下面是一个以估计值`x`开始，并不断迭代优化直到它不再变化的循环：

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

对于大多数`a`值来说，这样效果很好，但通常来说，测试`float`的相等是危险的。浮点数值只是近似正确：大部分有理数，如1/3，以及

无理数，如$\sqrt{2}$，都不能用`float`精确表示。

比起判断`x`和`y`是否精确相等，更安全的方式是利用内置函数`abs`来计算它们之间差值的绝对值，或者说量级：

```
if abs(y-x) < epsilon:
    break
```

这里`epsilon`的值是0.0000001，用来决定近似度是足够的。

7.6 算法

牛顿方法是算法 的一个例子：它是解决一类问题的机械化过程（在这个例子里，问题是计算平方根）。

要理解算法是什么，从一个算不上算法的东西开始可能更简单。在学习个位数相乘时，你可能背诵过乘法表。实际上，你记住了100个特别的答案。这种知识不算是算法。

但是，如果你比较“懒”，可能已经学会了一些小技巧来偷懒。例如，要计算n和9的乘积，你可以写下$n-1$作为十位数，$10-n$作为个位数。这个小技巧是计算任意个位数和9的乘积的通用方案。这算是一个算法！

相似地，你学过的进位加法、借位减法以及长除法都是算法。算法的特点之一是它们不需要任何聪明才智就能执行。它们是一个机械化的

过程，其中每一步都依照一组简单的规则接着上一步进行。

执行算法非常枯燥，但设计算法的过程却充满趣味和智力挑战，并且是计算机科学的一个核心部分。

一些人们自然而然、毫无困难或者下意识所做的事情，用算法表达却最为困难。理解自然语言是一个好例子。我们都能理解自然语言，但是至今为止还没有人能解释我们是怎么 做到的，至少没办法用算法解释。

7.7 调试

当你开始编写更大的程序时，常常会发现自己花费更多的时间用于调试。更多的代码意味着更多的出错机会，以及更多可能隐藏着bug的地方。

削减调试时间的方法之一是“二分调试”（debugging by bisection）。例如，如果你的程序有100行代码，如果每次检查一行，需要100步。

相反地，可以尝试把问题分成两半。找到程序的中点，或者接近那里的地方，找一个可以检验的中间结果。添加一个`print` 语句（或者其他的可以有检查效果的代码）并运行程序。

如果中点检验的结果是错误的，说明错误必然出现在程序的前半部分。如果是正确的，那错误则在程序的后半部分。

每进行一次这样的检查，就减少了一半需要检查的代码。经过6步之后（显然少于100步），就能够减少到一至两行代码，至少理论上如此。

实践中，常常很难确定“程序的中点”在哪里，并且并不总是能够检验它。通过数代码行数来确定中点显然没有意义。相反地，应当思考程序中哪些地方可能出错，哪些地方容易加上一个检查。然后选择一个你认为在其前后发生错误概率差不多的点进行检查。

7.8　术语表

重新赋值 （reassignment）：对一个已经存在的变量赋予一个新值。

更新 （update）：一种赋值操作，新值依赖于变量的旧值。

初始化 （initialization）：一种赋值操作，给变量一个初始的值，以后可以进行更新。

增量 （increment）：一种更新操作，增加变量的值（常常是加1）。

减量 （decrement）：一种更新操作，减少变量的值。

迭代 （iteration）：使用递归函数调用或者循环来重复执行一组语句。

无限循环 （infinite loop）：一个终止条件永远无法满足的循环。

算法（algorithm）：解决一类问题的通用过程。

7.9　练习

练习**7-1**

复制7.5节的循环并封装到一个名为square_root的 函数中，这个函数接收一个形参a 。选择一个合理的值x ，并返回a 的平方根的估计值。

要测试这个方法，可以编写一个名为test_square_root的 函数，打印下面这样的表格：

```
a      mysqrt(a)       math.sqrt(a)    diff
-      ---------       ------------    ----
1.0   1.0             1.0             0.0
2.0   1.41421356237   1.41421356237    2.22044604925e-16
3.0   1.73205080757   1.73205080757    0.0
4.0   2.0             2.0              0.0
5.0   2.2360679775    2.2360679775      0.0
6.0  2.44948974278   2.44948974278     0.0
7.0   2.64575131106   2.64575131106     0.0
8.0  2.82842712475   2.82842712475     4.4408920985e-16
9.0  3.0             3.0              0.0
```

第一列是一个数，*a*；第二列是数*a* 的平方根，使用mysqrt 函数计算；第三列是使用math.sqrt 计算出的平方根；第四列是两种计算结果的差值的绝对值。

练习**7-2**

内置函数`eval`接收一个字符串并使用Python解释器对它进行求值。例如：

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

编写一个函数`eval_loop`，迭代地提示用户，接收他们的输入并使用`eval`求值，并打印出结果。

它应当一直继续，直到用户输入`'done'`，并返回最后一个求值的表达式的结果。

练习**7-3**

数学家拉马努金（Srinivasa Ramanujan）找到了一个无限序列，可以用来生成π的数值近似值：

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

编写一个函数`estimate_pi`，使用这个公式计算并返回π的近似估计。它应当使用一个`while`循环来计算求和的每一项，直到最后一项的值小于`1e-15`（这是Python对10^{-15}的标记法）。你可以通过和`math.pi`比较来检查计算的结果。

解答：http://thinkpython2.com/code/pi.py。

第8章　字符串

字符串和整数、浮点数以及布尔类型都不同。字符串是一个序列（sequence），即它是一个由其他值组成的有序集合。本章中你将见到如何访问构成字符串的各个字符，并学到字符串类提供的一些方法。

8.1　字符串是一个序列

字符串是一个字符的序列 （sequence）。可以使用方括号操作符来访问字符串中单独的字符：

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第二个语句选择`fruit` 中的第1个字符，并将它赋值给`letter` 变量。

方括号中的表达式称为下标 （index）。下标表示想要序列中的哪一个字符（所以用index这个名称）。

但你可能发现得到的和预料不一样：

```
>>> letter
'a'
```

对大多数人来说，'banana' 的第一个字母是b ，而不是a 。但对计算机科学家来说，下标表示的是离字符串开头的偏移量，而第一个字母的偏移量是0。

```
>>> letter = fruit[0]
>>> letter
'b'
```

所以b 是'banana' 的第0个字母，a 是第1个，n 是第2个。

可以使用包括变量和操作符的表达式作为下标。

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

但下标的值必须是整数，否则你会得到：

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 len

len 是一个内置函数，返回字符串中字符的个数：

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

要获得字符串的最后一个字母，你可能会想这么写：

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

IndexError 出现的原因是'banana' 中没有下标为6的字母。因为我们是从0开始计算的，6个字母的下标是0到5。要获得最后一个字符，需要从length 里减1：

```
>>> last = fruit[length-1]
>>> last
'a'
```

或者，你可以使用负数下标。负数下标从字符串结尾处倒着数。表达式fruit[-1] 返回最后一个字母，表达式fruit[-2] 返回倒数第二

个字母，依此类推。

8.3 使用for 循环进行遍历

有很多计算都涉及对字符串每次处理一个字符的操作。它们常常从开头起，每次选择一个字符，对它做一些处理，再继续，直到结束。这种处理的模式，我们称为遍历（traversal）。编写遍历逻辑的方法之一是使用while 循环：

```
index = 0
while index < len(fruit):
      letter = fruit[index]
      print(letter)
      index = index + 1
```

这个循环遍历字符串，并将每个字符显示在单独的一行上。循环的结束条件是index < len(fruit)，所以当index 等于字符串的长度时，条件为假，循环体不被运行。最后访问的字符下标为len(fruit)-1，正好是字符串最后一个字符。

作为练习，写一个函数，接收一个字符串作为形参，并倒序显示它的字母，每个字母单独一行。

写遍历逻辑的另一个方式是使用for 循环：

```
for letter in fruit:
    print(letter)
```

每次迭代之中，字符串中的下一个字符会被赋值给变量`letter`。循环会继续直到没有剩余的字符为止。

下面的示例展示了如何利用字符串拼接（字符串加法）和一个`for`循环来生成字母序列（也就是，按字母顺序排序的序列）。在Robert McCloskey的书《为小鸭让路》（*Make Way for Ducklings*）中，小鸭们的名字是Jack、Kack、Lack、Mack、Nack、Ouack、Pack及Quack。下面的循环按顺序输出这些名字：

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

输出是：

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

当然那并不完全正确，因为“Ouack”和“Quack”拼写错了。作为练习，修改程序解决这个问题。

8.4　字符串切片

字符串中的一段称为一个切片 （slice）。选择一个切片和选择一个字符类似：

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

操作符[n:m]返回字符串从第n个字符到第m个字符的部分，包含第n个字符，但不包含第m个字符。这个行为有些违反直觉，但如果想象下标是指向字符之间 的位置，可以帮助我们理解它，如图8-1所示。

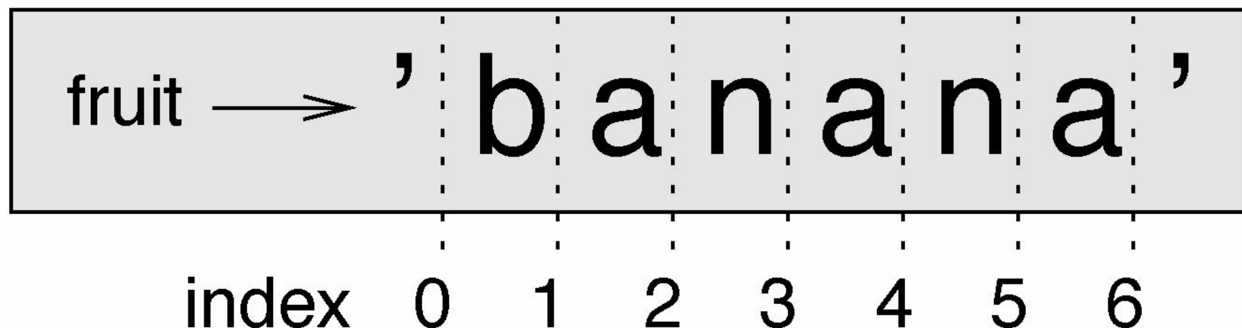


图8-1　切片的下标

如果省略掉第一个下标（冒号之前的那个），切片会从字符串开头

开始。如果省略掉第二个下标，切片会继续到字符串的结尾。

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

如果第一个下标大于或等于第二个下标，结果是空字符串 ，用两个引号表示：

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

空字符串不包含任何字符，长度为0，但除此之外，它和其他字符串一样。

继续本例，你认为`fruit[:]`表示什么？尝试一下看看结果。

8.5　字符串是不可变的

想要修改字符串的某个字符，你可能会想直接在赋值左侧使用`[]`操作符。例如：

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

这个例子里的“对象”（object）是字符串，而“项”（item）是指你想要赋值的那个字符。就现在来说，一个对象 和值是差不多的东西，但我们会在后面细谈它（参见10.10节）。

这个错误产生的原因是因为字符串是不可变 （immutable）的，也就是说，不能修改一个已经存在的字符串。你能做的最多是新建一个字符串，它和原来的字符串稍有不同：

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

这个例子使用新的首字符和`greeting` 的一个切片拼接起来。它对原来的字符串没有影响。

8.6　搜索

下面的这段函数是做什么的？

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
```

```
    return - 1
```

从某种意义上说，find 是[] 操作符的反面。和[] 操作符通过一个下标查找对应的字符不同，它根据一个字符查找其出现在字符串中的下标。如果没有找到字符，函数返回-1 。

这是我们第一次在循环内部看到return 语句。如果word[index] == letter ，函数直接跳出循环并立即返回。

如果字符没有出现在字符串中，程序正常退出循环，并返回-1 。

这种计算的模式——遍历一个序列，并当找到我们寻找的目标时返回——称为搜索 。

作为练习，修改find 函数，让它接收第3个形参，表示从word 的哪个下标开始搜索。

8.7 循环和计数

下面的代码计算字母a 在字符串中出现的次数：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

这个程序展示了另一种计算模式，称为计数器 。变量count 初始化为0，接着每次找到一个a 时计数器加1。当循环结束时，count 保存着结果——a 出现的总次数。

作为练习，将这段代码封装成函数count ，并泛化它以接收字符串和要计数的字母作为形参。

接着重写count 函数，不直接遍历字符串，而是使用前面一节中的3形参版本的find 函数。

8.8　字符串方法

字符串提供了很多完成各种操作的有用的方法。方法 和函数很相似——它接收形参并返回值——但语法有所不同。例如，方法upper 接收一个字符串，并返回一个全部字母都是大写的字符串。

和函数的语法upper(word) 不同，它使用方法的调用语法word.upper() 。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

这种句点表示法指定了方法的名称，以及方法应用到的字符串的名

称word 。空的括号表示这个方法没有任何参数。

方法的调用称为invocation [1]；在这个例子里，我们说我们在word 字符串上调用方法upper 。

实际上，字符串本来就有一个方法find ，和我们之前写的find 函数非常相似：

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

在这个例子中，我们在word 上调用find 方法，并传入要查找的字母作为实参。

实际上，find 方法比我们的函数更通用；它可以用来查找子字符串，而不仅仅是字符：

```
>>> word.find('na')
2
```

默认情况下，find 在字符串的开始启动，但它还可以接收第二个实参，表示从哪一个下标开始查找：

```
>>> word.find('na', 3)
4
```

这是可选参数的一个示例。find 还可以接收第三个实参，表示查找到哪个下标就结束：

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

这个搜索失败，因为b 并没有在字符串的下标1到2之间（不包括2）出现。find 在搜索时只搜索到第二个（但不包括第二个）下标为止，这使find 和切片操作符的行为一致。

8.9　操作符in

in 是一个布尔操作符，操作于两个字符串上，如果第一个是第二个的子串，则返回True ，否则返回False ：

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

例如，下面的函数打印出word1 中出现且出现在word2 中的所有字

母：

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

精心选择变量名称后，Python有时会读起来很像英语。可以这样读这个循环："for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter"。

下面是用这个函数比较单词apples和oranges的结果：

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 字符串比较

关系操作符也可以用在字符串上。检查两个字符串是否相等：

```
if word == 'banana':
    print('All right, bananas.')
```

其他的关系操作符在将单词按照字母顺序比较时有用：

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python处理大小写字母时和人处理时不一样。所有的大写字母都在小写字母之前。所以：

```
Your word, Pineapple, comes before banana.
```

处理这个问题的常用办法是先将字符串都转换为标准的形式，如都转换成全小写字母形式，再进行比较。如果你遇到一个武装着Pineapple的敌人需要保护自己时，请记住这个办法。

8.11 调试

当使用下标来遍历序列中的值时，要正确实现遍历的开端和结尾并不容易。下面是一个函数，能够比较两个单词，如果它们互为倒序，则返回`True`，但这个函数包含了两个错误：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
```

```
    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
return False
        i = i+1
        j = j-1

    return True
```

第一个if 语句检查两个单词是否长度相同。如果不同，我们就立即返回False ，否则在后面整个函数中，都可以认为两个单词是相同长度的。这是6.8节中讲到的守卫模式的一个实例。

i 和j 是下标：i 用于正向遍历word1 ，而j 用于反向遍历word2 。如果我们找到两个不匹配的字母，则可以立即返回False 。如果完成整个循环后所有的字母仍然都相等，则返回True 。

如果使用单词“pots”和“stop”来测试这个函数，我们会预期返回值是True ，但实际上会得到一个IndexError：

```
>>> is_reverse('pots', 'stop')
...
    File "reverse.py", line 15, in is_reverse
      if word1[i] != word2[j]:
IndexError: string index out of range
```

为了调试这类错误，第一步可以在发生错误的那行代码之前打印出

索引的值。

```
while j > 0:
    print i, j       # 在这里打印

    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

这样再一次运行程序时，能获得更多的信息：

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

第一次迭代时，`j` 的值是4，超出了`'pots'` 的范围。最后一个字符的下标是3，所以`j` 的初始值应该是`len(word2)-1` 。

如果修改这个错误并重新运行程序，会得到：

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

这回我们得到了正确的结果，但看起来循环只运行了3次，有些可疑。为了对具体发生了什么有更清晰的印象，可以画一个状态图。第一个迭代中，`is_reverse` 的帧显示在图8-2中。

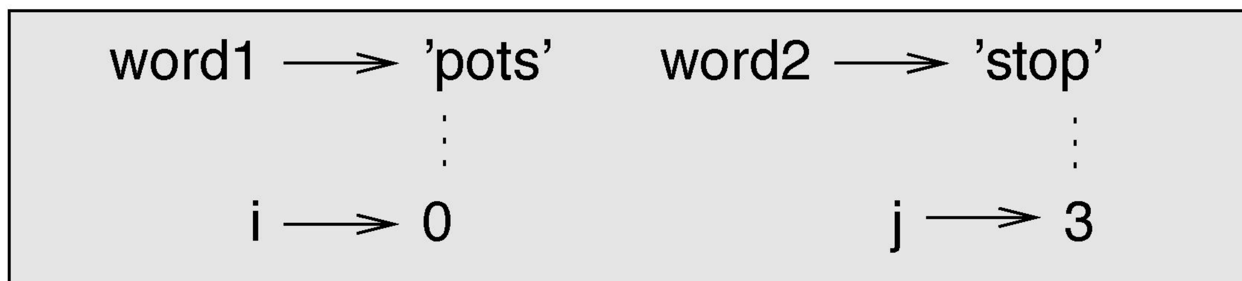


图8-2　状态图

我特意安排了帧中变量的位置，并使用虚线来显示`i` 和`j` 指向`word1` 和`word2` 中的字符。

从这个图开始，在纸上运行程序，每个迭代修改`i` 和`j` 的值。找到并修复这个函数的第二个错误。

8.12　术语表

对象（object）：变量可以引用的一种事物。就现在来说，可以把“对象”当作“值”来使用。

序列（sequence）：一个有序的值的集合，其中每个使用一个下标来定位。

项（item）：序列中的一个值。

下标（index）：用于在序列中选择元素的整数值。例如，可以用

于在字符串中选取字符。在Python中下标从0开始。

切片 （slice）：字符串的一部分，通过一个下标范围来定位。

空字符串 （empty string）：没有字符，长度为0的字符串，使用一对引号来表示。

不可变 （immutable）：序列的一种属性，表示它的元素是不可以改变的。

遍历 （traverse）：迭代访问序列中的每一个元素，并对每个元素进行相似的操作。

搜索 （search）：一种遍历的模式，当找到它想要的元素时停止。

计数器 （counter）：一种用来计数的变量，通常初始化为0，后来会递增。

方法调用 （invocation）：调用一个方法的语句。

可选参数 （optional argument）：函数或方法中，并不必须有的参数。

8.13　练习

练习**8-1**

在http://docs.python.org/3/library/stdtypes.html#string-methods阅读字符串方法的文档。你可能会想实验一下其中的一些方法，以确保自己理

解了它们的工作方式。strip 和replace 特别有用。

文档中使用了一种可能会引起困惑的语法。例如，find(sub[, start[, end]]) 中的方括号表示可选的参数。所以sub 是必需的，但是start 是可选的，并且如果使用了start ，则end 是可选的。

练习**8-2**

有一个字符串方法叫作count ，和我们之前在8.9节中展示的方法类似。阅读这个方法的文档，并写一个程序调用它来计算'banana' 中a 出现的次数。

练习**8-3**

字符串切片可以接受第三个下标用来指定“步长”，即相邻的字符之间的距离。步长为2，意思是切片每次取接下来第2个字符；步长3意思是每次取接下来第3个字符，等等。

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

步长为−1表示切片按照相反的方向访问字符串，所以切片[::-1] 会得到一个逆序的字符串。

使用这个特性来编写一个一行版本的is_palindrome 函数（见练习6-3）。

练习**8-4**

下面的几个函数目的 都是检查一个字符串是否包含小写字母，但至少有一个是错误的。对每个函数，描述一下这个函数到底做了什么（假设形参是一个字符串）。

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

练习**8-5**

凯撒密码（Caesar Cypher）是一个比较弱的加密形式，它涉及将单词中的每个字母“轮转”固定数量的位置。轮转一个字母意思是在字母表中移动它，如果需要，再从开头开始。所以‘A’轮转3个位置是‘D’，而‘Z’轮转一个位置是‘A’。

要对一个单词进行轮转操作，对其中每一个字母进行轮转即可。例如，“cheer”轮转7位的结果是“jolly”，而“melon”轮转−10位结果是“cubed”。在电影《2001太空漫游》中，舰载机器人叫作HAL，这个单词正是IBM轮转-1位的结果。

编写一个函数`rotate_word`，接收一个字符串以及一个整数作为参数，并返回一个新字符串，其中的字母按照给定的整数值“轮转”位置。

你可以使用内置函数`ord`，它能够将一个字符转换为数值编码，以及函数`chr`，它将数值编码转换为字符。字母表中的字母是按照字母顺序编码的，所以，例如：

```
>>> ord('c') - ord('a')
2
```

因为`'c'`在字母表中的下标是2。但是请注意：大写字母的数字编码是不同的。

因特网上有些可能冒犯人的笑话是用ROT13编码的。ROT13是轮转13位的凯撒密码。如果你不容易被冒犯，可以寻找一些并解码。

解答：http://thinkpython2.com/code/rotate.py。

---

[1] 普通函数的调用，称为`call`。——译者注

第9章　案例分析：文字游戏

本章介绍第二个案例分析，讲述的是通过搜索具有某种特性的单词来解决单词谜题这一话题。例如，我们会寻找英语单词中最长的回文单词，还会搜索那些其字母按照字母表顺序排列的单词。另外，我会介绍另一种程序开发计划：缩减问题规模，回归成之前解决过的问题。

9.1　读取单词列表

为本章的练习，我们需要准备一个英文单词列表。互联网上有很多可用的单词列表，但最适合我们的目标的单词列表，是由Grady Ward收集整理并作为Moby词典项目（参看http://wikipedia.org/wiki/Moby_Project）的一部分贡献给公共域的。它包含113 809个正式的填字游戏用词，即那些认为可以用于纵横填字游戏和其他类型文字游戏的单词。在Moby集合中，文件名是`113809of.fic`；可以从http://thinkpython.com/code/words. txt下载一个副本，但文件名是更简单的`words.txt`。

这个文件是纯文本，所以可以使用文本编辑器打开，也可以使用Python读入它。内置函数`open`接收文件名作为参数，并返回一个文件对象（file object），可以用来读取文件。

```
>>> fin = open('words.txt')
```

fin 是用来表示文件对象作为输入源时常用的名称。文件对象提供了几个方法用于读取内容，包括readline ，它会从文件里读入字符，直到获得换行符为止，并将读入的结果作为一个字符串返回：

```
>>> fin.readline()
'aa\r\n'
```

在这个特定的列表中，第一个单词是"aa"，它是一种火山熔岩。序列\r\n 表示两个空格字符，一个是回车，一个是换行，用于把这个单词和其他单词分隔开。

文件对象会记录它读到文件的哪个位置，因此如果再次调用readline ，会得到下一个单词：

```
>>> fin.readline()
'aah\r\n'
```

下一个单词是"aah"，也是一个完全合法的单词，所以别用奇怪的眼光看着我。或者，如果是那几个空白字符在干扰你，可以使用字符串的方法strip 去掉它们：

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

你也可以在`for`循环中使用文件对象。下面的代码读入`words.txt`并每行打印出一个单词：

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2 练习

在下一节里有这些练习的解答。在继续阅读解答之前，应当至少尝试一下每一个练习。

练习**9-1**

编写一个程序，读入`words.txt`并且打印出那些长度超过20个字符的单词（不算空白字符）。

练习**9-2**

1939年，Ernest Vincent Wright出版了一本5万字的小说*Gadsby*，这本书里没有包含字母“e”。因为“e”是英语中最常见的字母，所以这并不是件容易的事。

实际上，不使用这最常见的字母的话，仅仅是构建一条单独的构思也是很难的事情。开始时会很慢很艰难，但保持谨慎和长时间的训练，你可以渐渐掌握方法。

好吧，我先停下来。[1]

写一个函数has_no_e ，当给定的单词不包含字母“e”时，返回True 。

修改前一节练习中的代码，打印出不含“e”的单词，并计算这种单词在整个单词表中的百分比。

练习**9-3**

编写一个函数avoids ，接收一个单词，以及一个包含禁止字母的字符串，当单词不含任何禁止字母时，返回True 。

修改你的程序，提示用户输入包含禁止字母的字符串，并打印出不包含任意禁止字母的单词的个数。能不能找到一组5个禁止字母的组合，它们排除的单词最少？

练习**9-4**

编写一个名为uses_only 的函数，接收一个单词以及字母组成的字符串，当单词只由这些字母组成时返回True 。你可以造一个句子，其单词只由字母acefhlo 组成吗？除了“Hoe alfalfa”之外呢？

练习**9-5**

编写一个名为uses_all 的函数，接收一个单词以及由需要的字母组成的字符串，当单词中所有需要的字母都出现了至少一次时返回True 。有多少单词使用了所有的元音字母aeiou ？而aeiouy 呢？

练习**9-6**

编写一个名叫is_abecedarian 的函数，如果单词中的字母是按照字母表顺序排列的（两个重复字母也可以），则返回True 。有多少这样的单词？

9.3 搜索

前面一节的所有练习都有一个共同点；它们可以使用我们在8.6节中介绍的搜索模式来解决。最简单的例子是：

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

for 循环遍历单词word 中的字符。如果我们找到字母“e”，可以立即返回False ；否则只能继续下一个字母。如果正常退出了循环，则说明我们没有找到“e”，所以返回True 。

使用in 操作符，可以把这个函数写得更简洁。上面这个示例没有写得更简洁是因为想要展现搜索模式的逻辑。

avoids 是has_no_e 的更通用的版本，它们的结构相同：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

一旦发现一个禁止的字母，可以立即返回False ；如果运行到循环结束，则返回True 。

uses_only 函数也类似，只是它条件判断的意思是相反的：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

它接收的参数并不是一个禁止字母列表，而是一个可用字母列表available 。如果我们发现单词中遇到了并不属于available 的字母，则可以返回False 。

uses_all 函数也类似，但单词和字母列表的角色相反。

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

我们不再遍历单词`word`中的字母，而是循环遍历必需的单词列表`required`。如果单词列表中有任意字母没有出现在单词中，我们可以返回`False`。

如果你真的像计算机科学家那样思考的话，应该已经发现，`uses_all`实际上是已经解决的问题的一个特例，并且可以这么写：

```
def uses_all(word, required):
    return uses_only(required, word)
```

这是被称为将问题回归到已解决问题（reduction to a previously solved problem）的程序开发计划的一个例子。意即你需要识别出的当前问题是一个已经解决的问题的特例，从而可以直接利用现有的解决方案。

9.4　使用下标循环

在前面一节的例子中，我使用`for`循环进行遍历，因为只需要字符串中的字符，而不需要操作下标。

但对`is_abecedarian`函数我们需要比较相邻的字母，使用`for`循

环比较困难：

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

或者也可以使用递归：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

还有一个办法是使用while 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

循环开始于i=0， 并结束于i=len(word)-1 。每次迭代时，比较第i 个字符（可以看成是当前字符）和第i +1个字符（可以看成是下一个字符）。

如果下一个字符比当前字符小（即按照字母顺序在前），则我们发现了一个破坏字母顺序的断点，可以返回False 。

如果我们没有找到任何断点而结束循环，则这个单词通过了测试。为了说服自己循环是正确结束的，可以考虑像'flossy' 这样的例子。这个单词的长度是6，所以最后一次循环时i 是4，即是倒数第二个字符的下标。在最后一个循环中，会比较倒数第二个和最后一个字符，这正是我们所期待的。

下面是is_palindrome 函数（参考练习 6-3）的一个版本，它使用两个下标；一个从0开始递增；另一个从最后开始递减。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

或者，我们可以将其回归到已经解决的问题，可能这么写：

```
def is_palindrome(word):
    return is_reverse(word, word)
```

使用练习8-2中的`is_reverse`。

9.5　调试

测试程序很难。本章中的函数相对容易测试，因为可以简单地手动验证结果。即便如此，要选择一组可以测试到所有可能的错误的单词，也是很困难的，甚至是不可能的。

举`has_no_e`作为例子，有两个很明显的用例可以检测：包含“e”的单词应该返回`False`；不包含“e”的应当返回`True`。为这两种情况找到具体的单词没有问题。

但对每种情况来说，也存在一些不那么明显的具体情况。在所有包含“e”的单词中，你应当测试以“e”开头的单词，也应当测试以“e”结尾，以及“e”在单词中部的情况。你应当测试长单词、短单词及非常短的单词，如空字符串。空字符串是特殊情形（special case）的一个例子。特殊情形往往不那么明显，但又常常隐藏着错误。

除了自己生成的测试用例之外，还可以使用类似`words.txt`这样的单词表来测试你的程序。通过扫描输出，可能会发现错误，但请注意：你可能发现一种类型的错误（不应该被包含但却被包含的单词），但对另一种类型的则不能发现（应该被包含，但却没有出现的单词）。

总之，测试可以帮助你发现bug，但生成一组好的测试用例并不容易。而且，即使有好的测试用例，也无法确定程序是完全正确的。引用一个传奇计算机科学家的话：

程序测试可以用来显示bug的存在，但无法显示它们的缺席！（Program testing can be used to show the presence of bugs， but never to show their absence!）

——Edsger W.Dijkstra

9.6 术语表

文件对象 （file object）：用来表示一个打开的文件的值。

将问题回归到已解决问题 （reduction to a previously solved problem）：通过把问题表述为已经解决的某个问题的特例解决问题的一种方式。

特殊情形 （special case）：一种不典型或者不明显（因此更可能没有正确处理）的测试用例。

9.7 练习

练习**9-7**

本练习中的问题是基于广播节目《车迷天下》（Car Talk）中出现的一个谜题而设计的 （http://www.cartalk.com/content/puzzlers ）：

给我一个包含3组连续的成对字母的单词。我会给你几个几乎可以达到要求却还差一点儿的词作为例子。例如，单词committee，即c-o-m-m-i-t-t-e-e。除了i不满足条件外，这个单词是一个好例子。或者Mississippi：M-i-s-s-i- s-s-i-p-p-i。如果你能够拿掉其中的i，则它也符合要求。但确实有这么一个单词，并且就我所知，它可能是满足这个条件的唯一的单词。当然也可能存在500个，但我只能想到一个。它是什么呢？

编写一个程序来找到它。解答：http://thinkpython2.com/code/cartalk1.py。

练习**9-8**

下面是另一个《车迷天下》中的谜题（http://www.cartalk.com/content/puzzlers ）：

“有一天我正在高速公路上开车，碰巧注意到里程表。和大部分里程表一样，它显示6位整数的英里数。所以，例如我的车有300 000英里里程，则会看到3-0-0-0-0-0。

“那天我看到的里程数很有意思。我发现最后4位数是回文的；也就是说，它们不论是正序还是逆序地看都一样。例如，5-4-4-5是一个回文，所以我的里程表可能显示为3-1-5-4-4-5。

“1英里之后，后5位数组成一个回文。例如，它可以是3-6-5-4-5-6。再过1英里，6位数的中间4位是一个回文。而接下来，你准备好了吗？又1英里过去，所有的6位数都成了回文！

“问题是，我第一次看里程表时，它的示数是多少？”

编写一个Python程序，检测全部的6位数，并打印出可以满足上面这些要求的数字。解答：http://thinkpython2.com/code/cartalk2.py。

练习**9-9**

下面是另一个《车迷天下》的谜题，你可以使用一个搜索来解决（http://www.cartalk.com/ content/puzzlers ）：

“最近我去母亲家时，我发现自己的年龄的两位数正好是母亲的年龄的两位数的倒序。例如，如果她是73岁，我是37岁。我们好奇这种事情这些年来发生过几次，但很快我们的话题就偏转到其他地方，所以没有得到答案。

“我回家后，发现我们的年龄互为倒序的事情至今为止发生过6次。我还发现，如果顺利的话接下来几年还会再遇到一次，并且在那之后如果我们真的很幸运，还能再遇到一次。换句话说，它总共可能发生8次。所以问题是，我现在年龄多大？”

编写一个Python程序，为这个谜题搜索答案。提示：你可能会发现字符串方法`zfill` 有用。

解答：http://thinkpython2.com/code/cartalk3.py。

---

[1] 作者在上一段话中模仿了Gadsby的风格，不使用字母“e”，所以说话的风格很怪。因此到这里，他就说“All right，I’ll stop now”，意思是停

止这种怪异风格的描述。但这个意思无法在译文中表达。上一段话的英文原文是：“In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow at first, but with caution and hours of training you can gradually gain facility. ”——译者注

第10章　列表

本章介绍Python语言最有用的内置类型之一：列表。你还能学到更多关于对象的知识，以及同一个对象有两个或更多变量时会发生什么。

10.1　列表是一个序列

和字符串相似，列表 （list）是值的序列。在字符串中，这些值是字符；在列表中，它可以是任何类型。列表中的值称为元素（element），有时也称为列表项 （item）。

创建一个列表有好几种方式。其中最简单的方式是使用方括号（[与] ）将元素括起来。

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

第一个例子是4个整数的列表。第二个例子是3个字符串的列表。列表中的元素并不一定非得是同一类型的。下面的列表包含了一个字符串、一个浮点数、一个整数及（瞧！ ）另一个列表：

```
['spam', 2.0, 5, [10, 20]]
```

列表中出现的列表是嵌套的（nested）。

不包含任何元素的列表称为空列表，可以使用空方括号[]来创建空列表。

如你所预料的，列表可以赋值给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10.2　列表是可变的

访问列表元素的语法和访问字符串中字符的语法是一样的——使用方括号操作符。方括号中的表达式指定下标。请记得下标是从0开始的：

```
>>> cheeses[0]
'Cheddar'
```

和字符串不同的是，列表是可变的。当方括号操作符出现在赋值语句的左侧时，它用于指定列表中哪个元素会被赋值。

```
>>> numbers = [42, 123]
```

```
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

numbers 的第1位元素，原先的值是123，现在是5了。

图10-1显示了cheeses 、numbers 和empty 的状态图。

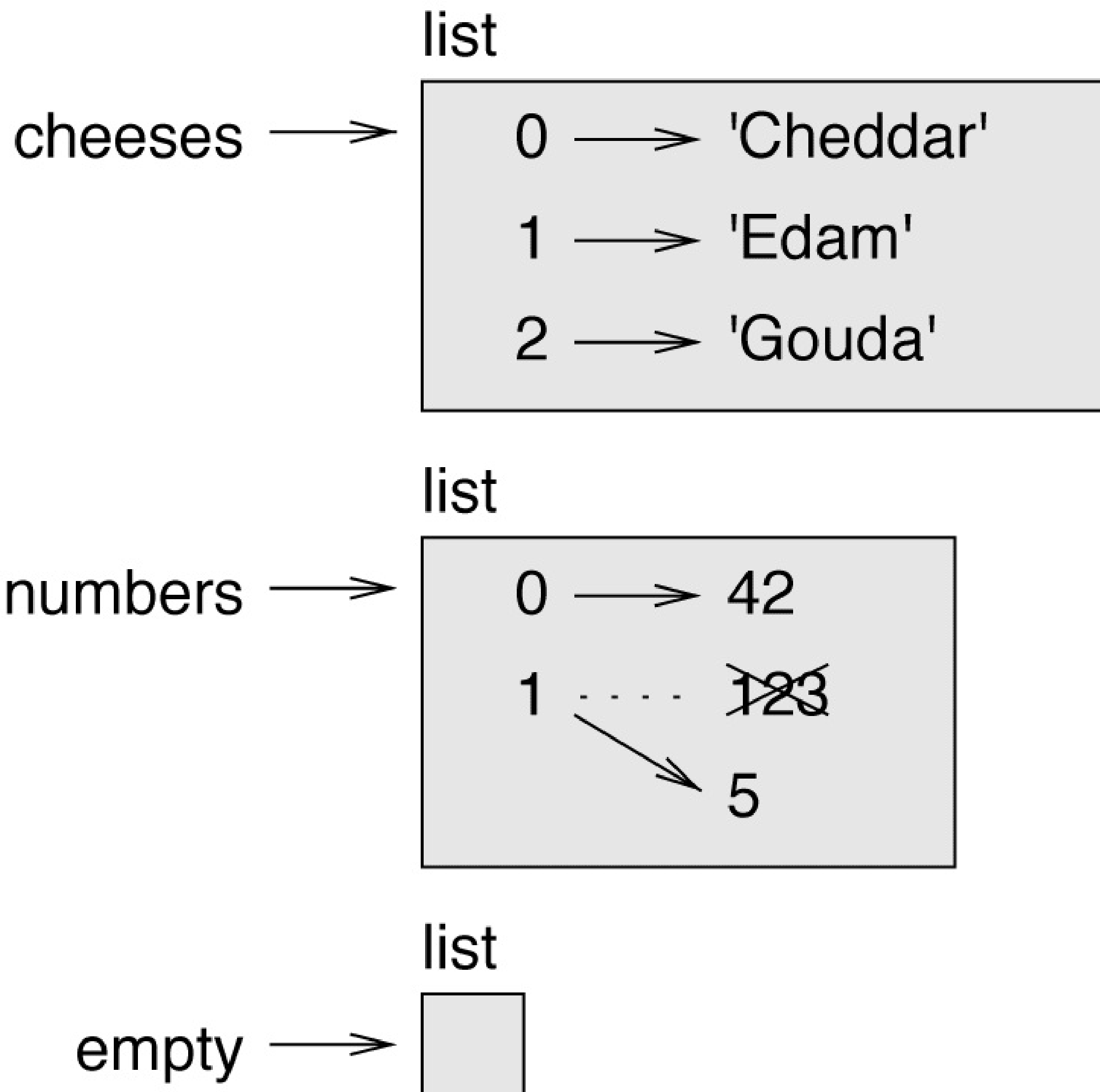


图10-1　状态图

在图10-1中，外面写有“list”的图框表示列表，里面显示的是列表中的元素。`cheeses` 变量引用着一个列表，包含3个元素，下标分别是0、1和2。`numbers` 包含两个元素；本图显示了其第二个元素从123重新赋值为5的过程。`empty` 引用一个没有任何元素的空列表。

列表下标和字符串下标工作方式相同。

- 任何整型的表达式都可以用作下标。
- 如果尝试读写一个并不存在的元素，则会得到`IndexError` 。
- 如果下标是负数，则从列表的结尾处反过来数下标访问。

`in` 操作符也可以用于列表。

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3　遍历一个列表

遍历一个列表元素的最常见方式是使用`for` 循环。语法和字符串的遍历相同：

```
for cheese in cheeses:
    print(cheese)
```

在只需要读取列表的元素本身时，这样的遍历方式很好。但如果需要写入或者更新元素时，则需要下标。一个常见的方式是使用内置函数`range` 和`len` ：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

这个循环遍历列表，并更新每个元素。`len` 返回列表中元素的个数。`range` 返回一个下标的列表，从0到$n-1$，其中n 是列表的长度。每次迭代时，`i` 获得下一个元素的下标。循环体中的赋值语句使用`i` 来读取元素的旧值并赋值为新值。

在空列表上使用`for` 循环，则循环体从不会被运行：

```
for x in []:
    print('This never happens.')
```

虽然列表可以包含其他的列表，嵌套的列表仍然被看作一个单独的元素。下面的列表长度是4：

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4 列表操作

+操作符可以拼接列表：

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

*操作符重复一个列表多次：

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

第一个例子重复列表`[0]`四次。第二个例子重复列表`[1, 2, 3]`三次。

10.5 列表切片

切片操作符也可以用于列表：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
```

```
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

如果省略掉第一个下标，则切片从列表开头开始。如果省略掉第二个下标，则切片至列表结尾结束。如果两个下标都省略，则切片就是整个列表的副本。

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

因为列表是可变的，所以在对列表进行修改操作之前，复制一份是很有用的。

如果切片操作符出现在赋值语句的左侧，则可以更新多个元素：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6　列表方法

Python为列表提供了不少操作方法。例如，append 可以在列表尾部添加新的元素：

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

extend 方法接收一个列表作为参数，并将其所有的元素附加到列表中：

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

这个例子中t2 没有被修改。

sort 方法将列表中的元素从低到高重新排列：

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

列表的大多数方法全是无返回值的。它们修改列表，并返回None。如果不小心写了t = t.sort()，你可能对结果感到很失望。

10.7 映射、过滤和化简

如果想把列表中所有的元素加起来，可以使用下面这样的循环：

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

total 被初始化为0。每次循环中，x 获取列表中的一个元素。+= 操作符为更新变量提供了一个简洁的方式。这个增加赋值语句：

```
total += x
```

等价于：

```
total = total + x
```

随着循环的运行，total 会累积列表中的值的和；这样使用一个变

量有时称为累加器 （accumulator）。

对列表元素累加是如此常见的操作，以至于Python提供了一个内置函数`sum`：

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

类似这样，将一个序列的元素值合起来到一个单独的变量的操作，有时称为化简 （reduce）。

有时候你想要在遍历一个列表的同时构建另一个列表。例如，下面的函数接收一个字符串列表，并返回一个新列表，其元素是大写的字符串：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` 初始化为一个空列表；每次循环，我们给它附加一个元素。所以`res` 也是一种累加器。

像`capitalize_all` 这样的操作，有时被称为映射 （map），因为它将一个函数（在这个例子里是`capitalize` 方法）“映射”到一个序列

的每个元素上。

另一个常见的操作是选择列表中的某些元素，并返回一个子列表。例如，下面的函数接收一个字符串列表，并返回那些只包含大写字母的字符串：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` 是一个字符串方法，当字符串中只包含大写字母时返回 `True` 。

类似`only_upper` 这样的操作称为过滤 （filter），因为它选择列表中的某些元素，并过滤掉其他的元素。

列表的绝大多数常用操作都可以用映射、过滤和化简的组合来表达。

10.8　删除元素

从列表中删除元素，有多种方法。如果知道元素的下标，可以使用`pop` ：

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
```

```
>>> t
['a', 'c']
>>> x
'b'
```

pop 修改列表，并返回被删除掉的值。如果不提供下标，它会删除并返回最后一个元素。

如果不需要使用删除的值，可以使用del 操作符：

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

如果知道要删除的元素（而不是下标），则可以使用remove ：

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

remove 方法的返回值是None 。

若要删除多个元素，可以使用del 和切片下标：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
>>> del t[1:5]
>>> t
['a', 'f']
```

和通常一样，切片会选择所有的元素，直到第二个下标（并不包含）。

10.9　列表和字符串

字符串是字符的序列，而列表是值的序列，但字符的列表和字符串并不相同。若要将一个字符串转换为一个字符的列表，可以使用函数`list`：

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

由于`list`是内置函数的名称，所以应当尽量避免使用它作为变量名称。我也避免使用`l`，因为它看起来太像数字`1`了。因而我使用`t`。

`list`函数会将字符串拆成单个的字母。如果想要将字符串拆成单词，可以使用`split`方法：

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
```

```
['pining', 'for', 'the', 'fjords']
```

split 还接收一个可选的形参，称为分隔符 （delimiter），用于指定用哪个字符来分隔单词。下面的例子中使用连字符（- ）作为分隔符：

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

join 是split 的逆操作。它接收字符串列表，并拼接每个元素。join 是字符串的方法，所以必须在分隔符上调用它，并传入列表作为实参：

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

在这个例子里，分隔符是空格，所以join 会在每个单词之间放一个空格。若想不用空格直接连接字符串，可以使用空字符串'' 作为分隔符。

10.10　对象和值

如果我们运行下面的赋值语句：

```
a = 'banana'
b = 'banana'
```

我们知道a 和b 都是一个字符串的引用。但我们不知道它们是否指向同一个 字符串。有两种可能的状态，如图10-2所示。

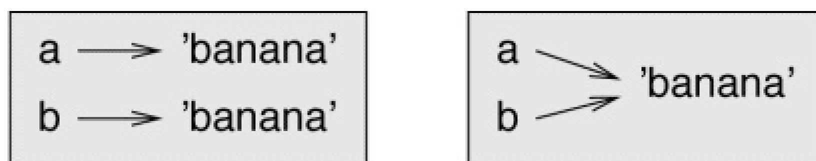


图10-2　状态图

一种可能是，a 和b 引用着不同的对象，它们的值相同。另一种情况下，它们指向同一个对象。

要检查两个变量是否引用同一个对象，可以使用is 操作符。

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

在这个例子里，Python只建立了一个字符串对象，而a 和b 都引用

它。

但当你新建两个列表时，会得到两个对象：

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

所以状态图如图10-3所示。

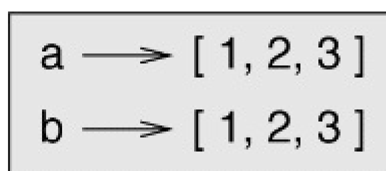


图10-3　状态图

在这个例子里我们会说这两个列表是相等的（equivalent），因为它们有相同的元素，但它们不是相同的（identical），因为它们并不是同一个对象。如果两个对象相同，则必然也相等，但如果两个对象相等，并不一定相同。

到目前为止，我们都不加区分地使用“对象”和“值”，但更精确的说法是对象有一个值。如果求值`[1,2,3]`，会得到一个列表对象，它的值是一个整数的序列。如果另一个列表包含相同的元素，我们说它有相同的值，但它们不是同一个对象。

10.11　别名

如果a 引用一个对象，而你赋值b = a ，则两个变量都会引用同一个对象：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

这里的状态图如图10-4所示。

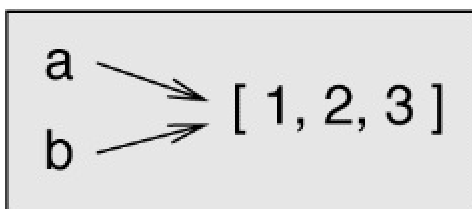


图10-4　状态图

变量和对象之间的关联关系称为引用 （reference）。在这个例子里，有两个指向同一对象的引用。

当一个对象有多个引用，并且引用有不同的名称时，我们说这个对象有别名 （aliased）。

如果有别名的对象是可变的，则对一个别名的修改会影响另一个：

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

虽然这种行为可能很有用，但它也容易导致错误。通常来说，当处理可变对象时，避免使用别名会更加安全。

对于字符串这样的不可变对象，别名则不会带来问题。在下面的例子中：

```
a = 'banana'
b = 'banana'
```

不论a 和b 是否引用同一个字符串，都不会有什么区别。

10.12 列表参数

当你将一个列表传递给函数中，函数会得到列表的一个引用。如果函数中修改了列表，则调用者也会看到这个修改。例如，delete_head 函数删除列表中的第一个元素：

```
def delete_head(t):
    del t[0]
```

下面使用它：

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

参数t 和变量letters 是同一个对象的别名。栈图如图 10-5所示。

因为列表被两个帧共享，所以我将它画在中间。

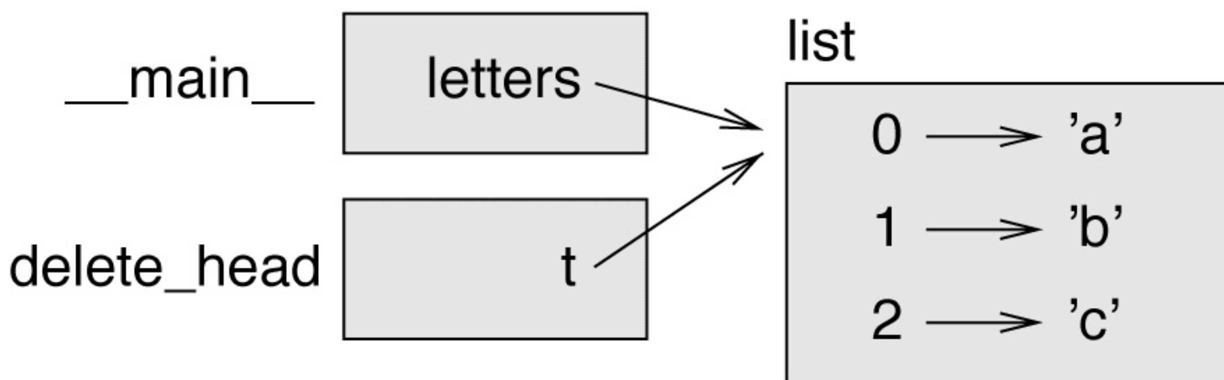


图10-5　栈图

区分修改列表的操作和新建列表的操作十分重要。例如，append 方法修改列表，但是+ 操作符新建一个列表：

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append 修改列表，返回None 。

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
```

```
>>> t3
[1, 2, 3, 4]
>>> t1
```

操作符+创建一个新列表，而原始的列表并不改变。

这个区别，在编写希望修改列表的函数时十分重要。例如，下面的函数并不会删除列表的开头：

```
def bad_delete_head(t):
    t = t[1:]                              # 错!
```

切片操作会新建一个列表，而赋值操作会让 t 引用指向这个新的列表，但这些操作对调用者没有影响。

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

在bad_delete_head 的开头，t 和t4 指向同一个列表。在函数最后，t 指向了一个新的列表，但t4 仍然指向原先的那个没有改变的列表。

另外一种方法是编写函数创建和返回一个新的列表。例如，tail

返回除了第一个以外所有的元素的列表：

```
def tail(t):
    return t[1:]
```

这个函数不会修改原始列表。下面的代码展示如何使用它：

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 调试

对列表（以及其他可变对象）的不慎使用，可能会导致长时间的调试。下面介绍一些常见的陷阱，以及如何避免它们。

1．大部分列表方法都是修改参数并返回None 的。这和字符串的方法正相反，字符串方法新建一个字符串，并留着原始的字符串不动。

如果你习惯于写下面这样的字符串代码：

```
word = word.strip()
```

则容易倾向于这么写列表代码：

```
t = t.sort()              # 错!
```

因为sort 返回None ，接下来对t 进行的操作很可能会失败。

在使用列表方法和操作符之前，应当仔细阅读文档，并在交互模式中测试它们。

2．选择一种风格，并坚持不变。

列表的问题之一是同样的事情有太多种可用的做法。例如，要从列表中删除一个元素，可以使用pop 、remove 、del 或者甚至是切片赋值。

要添加一个元素，可以使用append 方法或者+ 操作符。假设t 是一个列表，x 是一个列表元素，下面的操作是正确的：

```
t.append(x)
t = t + [x]
t += [x]
```

而下面的操作是错误的：

```
t.append([x])          # 错!
t = t.append(x)        # 错!
t + [x]                # 错!
```

```
t = t + x             # 错!
```

在交互模式中试验这些例子，确保你明白它们的运行细节。注意只有最后一个会导致运行时错误；其他的3个都是合法的，但是它们的结果不正确。

3．通过复制来避免别名。

如果想要使用类似sort 的方法来修改参数，但又需要保留原先的列表，可以复制一个副本：

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

在这个例子里也可以使用内置函数sorted ，它会返回一个新的排好序的列表，并且留着原先的列表不动。

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14 术语表

列表（list）：值的序列。

元素（element）：列表（或其他序列）中的一个值，也称为列表项。

嵌套列表（nested list）：作为其他列表的元素的列表。

累加器（accumulator）：在循环中用于加和或者累积某个结果的变量。

增加赋值（augmented assignment）：使用类似+=操作符来更新变量值的语句。

化简（reduce）：一种处理模式，遍历一个序列，并将元素的值累积起来计算为一个单独的结果。

映射（map）：一种处理模式，遍历一个序列，对每个元素进行操作。

过滤（filter）：一种处理模式，遍历列表，并选择满足某种条件的元素。

对象（object）：变量可以引用的东西。对象有类型和值。

相等（equivalent）：拥有相同的值。

相同 （identical）： 是同一个对象（并且也意味着相等）。

引用 （reference）： 变量和它的值之间的关联。

别名 （aliasing）： 多个变量同时引用一个对象的情况。

分隔符 （delimiter）： 用于分隔字符串的一个字符或字符串。

10.15　练习

你可以从http://thinkpython2.com/code/list_exercises.py下载这些练习的解答。

练习**10-1**

编写一个名为`nested_sum`的 函数，接收一个由内嵌的整数列表组成的列表作为形参，并将内嵌列表中的值全部加起来。例如：

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

练习**10-2**

编写一个名为`cumsum` 的函数，接收一个数字的列表，返回累计和；也就是说，返回一个新的列表，其中第i 个元素是原先列表的前$i+1$个元素的和。例如：

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

练习**10-3**

编写一个函数middle，接收一个列表作为形参，并返回一个新列表，包含除了第一个和最后一个元素之外的所有元素。例如：

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

练习**10-4**

编写一个名为chop的 函数，接收一个列表，修改它，删除它的第一个和最后一个元素，并返回None 。例如：

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

练习**10-5**

编写一个名为is_sorted的 函数，接收一个列表作为形参，并当列表是按照升序排好序的时候返回True ， 否则返回False 。

例如：

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

练习**10-6**

两个单词，如果重新排列其中一个的字母可以得到另一个，它们互为回文（anagram）。编写一个名为is_anagram的 函数，接收两个字符串，当它们互为回文时返回True 。

练习**10-7**

编写一个名为has_duplicates的 函数接收一个列表，当其中任何一个元素出现多于一次时返回True 。它不应当修改原始列表。

练习**10-8**

这个练习谈的是所谓的生日悖论，你可以在http://en.wikipedia.org/wiki/Birthday_paradox阅读相关资料。

如果你的班级中有23个学生，那么其中有两人生日相同的概率有多大？你可以通过随机生成23个生日的样本并检查是否有相同的匹配来估

计这个概率。提示：可以使用`random`模块中的`randint`函数来生成随机生日。

你可以从http://thinkpython2.com/code/birthday.py下载解答。

练习**10-9**

编写一个函数，读取文件`words.txt`，并构建一个列表，每个元素是一个单词。给这个函数编写两个版本，其中一个使用`append`方法，另一个使用`t = t + [x]`的用法。哪一个运行时间更长？为什么？

解答：http://thinkpython2.com/code/wordlist.py。

练习**10-10**

要检查一个单词是否出现在单词列表中，可以使用`in`操作符，但由于它需要按顺序搜索所有单词，可能会比较慢。

因为单词是按字母顺序排列的，我们可以使用二分查找（也叫作二分搜索）来加快速度。二分查找的过程类似于在字典中查找单词。从中间开始，检查需要找的单词是不是在列表中间出现的单词之前，如果是，则继续用同样的方法搜索前半部分。否则搜索后半部分。

不论哪种情形，都将搜索空间减小了一半。如果单词列表有113,809个单词，那么大概耗费17步就能找到单词，或者确认它不在列表之中。

编写一个函数`in_bisect`，接收一个排好序的列表，以及一个目标

值，当目标值在列表之中，返回其下标，否则返回`None` 。

或者你可以阅读`bisect` 模块的文档，并使用它！

解答：http://thinkpython.com2/code/inlist.py。

练习**10-11**

两个单词，如果其中一个是另一个的反向序列，则称它们为“反向对”。编写一个程序找到单词表中出现的全部反向对。

解答：http://thinkpython2.com/code/reverse_pair.py。

练习**10-12**

两个单词，如果从每个单词中交错取出一个字母可以组成一个新的单词，我们称它们为“互锁”（interlocking）。例如，“shoe”和 “cold”可以互锁组成单词“schooled”。

解答：http://thinkpython2.com/code/interlock.py。鸣谢：这个练习启发自http://puzzlers.org的一个示例。

1．编写一个程序找到所有互锁的词。提示：不要穷举所有的词对！

2．能不能找到“三互锁”的单词？也就是，从第一、第二或者第三个字母开始，每第三个字母合起来可以形成一个单词。

第11章　字典

本章介绍另一种内置类型：字典。字典是Python最好的语言特性之一，它是很多高效而优雅的算法的基本构建块。

11.1　字典是一种映射

字典 类似于列表，但更加通用。在列表中，下标必须是整数；而在字典中，下标（几乎）可以是任意类型。

字典包含下标（称为键 ）集合和值集合。每个键都与一个值关联。键和值之间的关联被称为键值对 （key-value pair），或者有时称为一项 （item）。

用数学语言来描述，字典体现了键到值的映射 ，所以可以说每个键“映射”到一个值。作为示例，我们构建一个字典，将英语单词映射到西班牙语上，所以键和值的类型都是字符串。

函数`dict` 新建一个不包含任何项的字典。因为`dict` 是内置函数的名称，应当避免使用它作为变量名。

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

这里花括号{} 表示一个空的字典。想要给字典添加新项，可以使用方括号操作符：

```
>>> eng2sp['one'] = 'uno'
```

这一行代码创建一个新项，将键'one' 映射到值'uno' 上。如果我们再次打印这个字典，可以看到一个键值对，以冒号分隔：

```
>>> eng2sp
{'one': 'uno'}
```

这种输出格式也同样是输入的格式。例如，可以创建一个包含3项的新字典：

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

但如果你打印eng2sp ，可能会觉得奇怪：

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

字典中键值对的顺序可能并不相同。如果你在自己的电脑上输入相同的示例，可能会得到另一个不同的结果。总之，字典中各项的顺序是不可预料的。

但这并不是问题，因为字典的元素从来不使用整数下标进行查找。相对地，它使用键来查找对应的值：

```
>>> eng2sp['two']
'dos'
```

如果键`'two'` 总是映射到值`'dos'` 上，那么各项的顺序其实并不重要。

如果一个键并不在字典之中，会得到一个异常：

```
>>> eng2sp['four']
KeyError: 'four'
```

`len` 函数可以用在字典上，它返回键值对的数量：

```
>>> len(eng2sp)
3
```

`in` 操作符也可以用在字典上，它告诉你一个值是不是字典中的键

（是字典中的值则不算）。

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

若要查看一个值是不是出现在字典的值中，可以使用方法`values`，它会返回一个值集合，并可以应用`in`操作符：

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

`in`操作符对列表和字典使用不同的算法实现。对于列表，它按顺序搜索列表的元素，如8.6节所示。当列表变长时，搜索时间会随之变长。

而对于字典，Python使用一个称为散列表（hashtable）的算法。它有一个值得注意的特点：不管字典中有多少项，`in`操作符花费的时间都差不多。我会在21.4节中解释其中的原因，但最好再多读几章，这样才可能看懂解释的内容。

11.2　使用字典作为计数器集合

假设给定一个字符串，你想要计算每个字母出现的次数。有几种可能的实现方法：

1．你可以创建26个变量，每个变量对应字母表上的一个字母。接着遍历字符串，对每一个字符，增加对应的计数器。你可能需要使用一个链式条件判断。

2．你可以创建一个包含26个元素的列表。接着可以将每个字符转换为一个数字（使用内置函数ord ），使用这个数字作为列表的下标，并增加对应的计数器。

3．你可以建立一个字典，以字符作为键，以计数器作为相应的值。第一次遇到某个字符时，在字典中添加对应的项。之后可以增加一个已经存在的项的值。

这几种方案进行相同的计算，但实现计算的方式不一样。

实现 （implementation）是进行某种计算的一个具体方式；有的实现比其他的更好。例如，字典实现的优势之一是我们并不需要预先知道字符串中可能出现哪些字母，因而只需为真正出现过的字母分配空间。

下面是这个实现的代码：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

这个函数的名称是直方图（histogram），它是一个统计学术语，表示一个计数器（或者说频率）的集合。

函数的第一行创建一个空的字典。for 循环遍历字符串。每次迭代中，如果字符c 不在字典中，我们就创建一个新项，其键是c ，其值初始化为1（因为我们已经见到这个字符一次了）。如果c 已经在字典之中，我们增加d[c] 。

下面是这个函数的使用方式：

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

这个直方图显示，字母'a' 和'b' 出现了1次；'o' 出现了两次，依此类推。

字典有一个方法get ，接收一个键以及一个默认值。如果键出现在字典中，get 返回对应的值；否则它返回默认值。例如：

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

作为练习，使用get 将histogram 写得更紧凑一些。你应当可以消除掉if 语句。

11.3　循环和字典

如果在for 循环中使用字典，会遍历字典的键。例如，print_hist 函数打印字典的每一个键以及对应的值：

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

下面是这个函数输出的样子：

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

同样地，键的出现没有特定的顺序。要按顺序遍历所有键，可以使用内置函数sorted：

```
>>> for key in sorted(h)
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4 反向查找

给定一个字典d 和键k ，找到对应的值v = d[k] 非常容易。这个操作称为查找 （lookup）。

但是如果有v ，而想找到k 时怎么办？这里有两个问题：首先，可能存在多个键映射到同一个值v 上。随不同的应用场景，也许可以挑其中一个，或者也许需要建立一个列表来保存所有的键。其次，并没有可以进行反向查找 的简单语法，你需要使用搜索。

下面是一个函数，接收一个值，并返回映射到该值的第一个键：

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

这个函数是搜索模式的又一个示例。但它使用了一个我们还没见过

的语言特性，raise 语句。 **raise** 语句 会生成一个异常；在这个例子里它生成一个LookupError ，这是一个内置异常，通常用来表示查找操作失败。

如果我们到达了循环的结尾，就意味着v 在字典中没有作为值出现过，所以我们抛出一个异常。

下面的例子展示了一个成功的反向查找：

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> k
'r'
```

以及一个不成功的反向查找：

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "<stdin>", line 5, in reverse_lookup
LookupError
```

当你自己抛出异常时，效果和Python抛出异常是一样的：它会打印出一个回溯和一个错误信息。

raise 语句也可以接收一个可选的参数用来详细描述错误。例如：

```
>>> raise LookupError('value does not appear in the dictionary')
```

```
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

反向查找远远慢于正向查找；如果频繁这么做，或者字典非常大时，会对程序的性能有很大影响。

11.5　字典和列表

列表可以在字典中以值的形式出现。例如，如果你遇到一个将字母映射到频率的字典，可能会想要反转它；也就是说，建立一个字典，将频率映射到字母上。因为可能出现多个字母频率相同的情况，在反转的字典中，每项的值应当是字母的列表。

这里是一个反转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

每次循环中，key 从d 中获得一个键，而val 获得相应的值。如果 val 不在inverse 字典中，意味着我们还没有见到过它，所以新建一个

项，并将它初始化为一个单件 （singleton，即只包含一个元素的列表）。否则我们已经见过这个值了，因此将相应的键附加到列表末尾。

下面是一个示例：

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

图11-1是显示`hist` 和`inverse` 的状态图。字典使用一个上方标明`dict` 的图框表示，内部包含键值对。如果值是整数、浮点数或字符串，我会把它们画到图框内，但我常常会将列表画在图框之外，以便保持状态图的简洁。

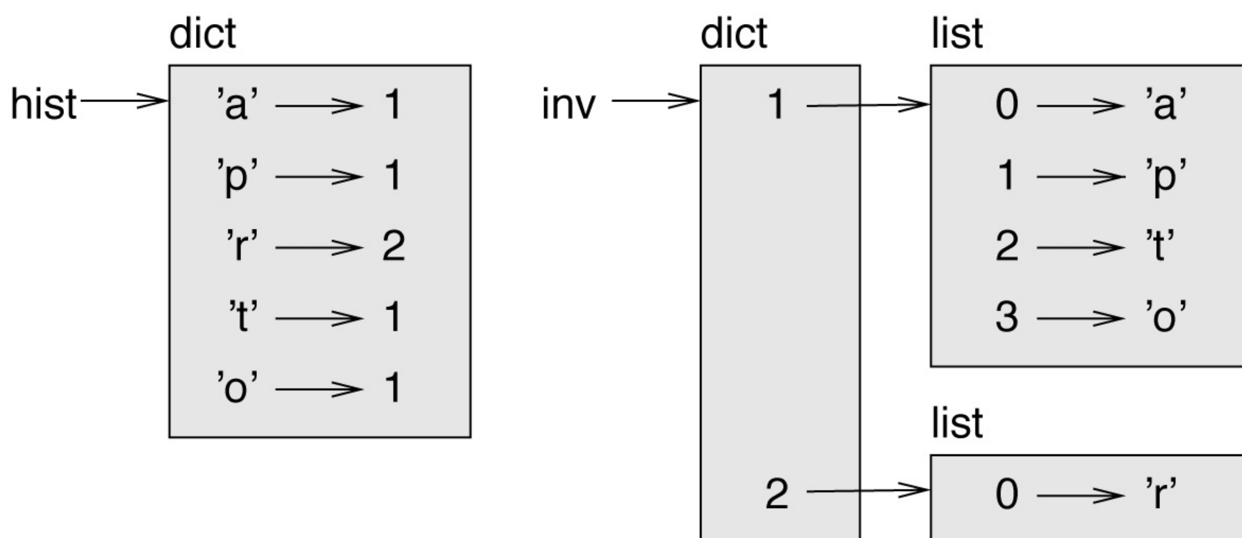


图11-1　状态图

如本例所示，列表可以用作字典的值，但它们不能用作键。如果尝试的话，会得到如下的结果：

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
     File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

之前我提到过字典是通过散列表的方式实现的，这意味着键必须是可散列（hashable）的。

散列 是一个函数，接收（任意类型）的值并返回一个整数。字典使用这些被称为散列值的整数来保存和查找键值对。

这套系统当键不可变时，可以正确工作。但如果像列表这样，键是可变的话，则会有不好的事情发生。例如，新建一个键值对时，Python 将键进行散列并存储到对应的地方。如果修改了键并再次散列，它会指向一个不同的地方。在那种情况下，会导致同一个键有两个条目，或者可能找不到某个键。不论如何，字典将无法正确工作。

因此键必须是可散列的，而类似列表这样的可变类型是不可散列的。绕过这种限制的最简单办法是使用元组，下一章会有详细介绍。

因为字典是可变的，它也不能用作键，但它可以 用作字典的值。

11.6　备忘

如果你尝试过6.7节中的`fibonacci`函数，可能会注意到，提供的参数越大，函数运行的时间越长，并且运行时间增长很快。

为了明白为什么会这样，考虑图11-2，它展示了`fibonocci`函数n=4时的调用图 。

调用图显示了一组函数帧，并用箭头将函数的帧和它调用的函数帧连接起来。在图的顶端，n=4 的`fibonacci`调用了n=3 和n=2 的`fibonacci` 。同样地，n=3 的`fibonacci`调用了n=2 和n=1 的`fibonacci` 。依此类推。

数一下`fibonacci(0)`和`fibonacci(1)`被调用了多少次。这是本问题的一个很低效的解决方案，而且当参数变大时，事情会变得更糟。

一个解决办法是记录已经计算过的值，并将它们保存在一个字典中。将之前计算的值保存起来以便后面使用的方法称为备忘（memo）。下面是一个使用了备忘的`fibonacci`版本：

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

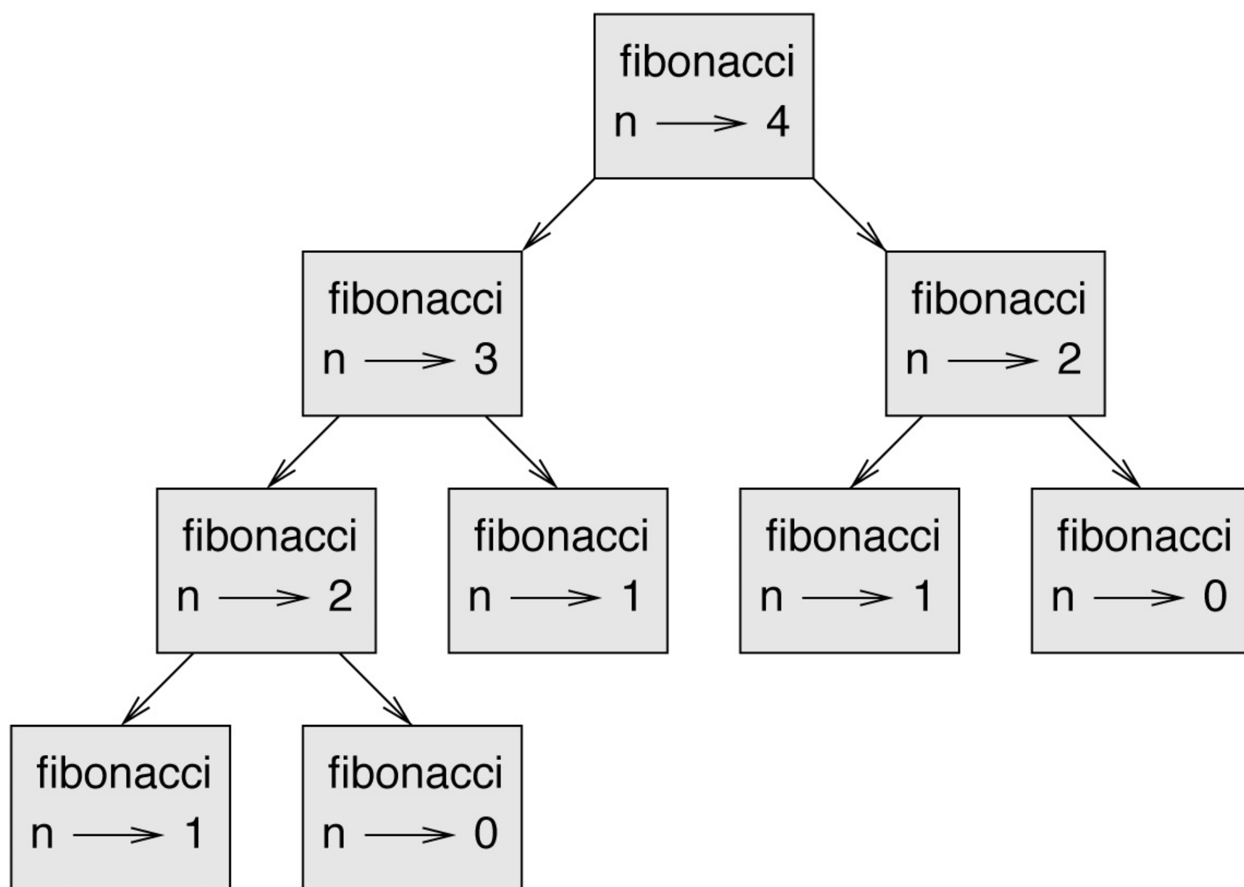


图11-2　调用图

`known` 是一个用来记录我们已知的Fibonacci数的字典。开始时它有两项：0映射到0，以及1映射到1。

每当`fibonacci` 被调用时，它会先检查`known` 。如果结果已经存在，则可以立即返回。如果不存在，它需要计算这个新值，将其添加进字典，并返回。

如果你运行fibonacci的这个版本，并将其与原始版本进行比较，你会发现，这个版本快得多。

11.7　全局变量

在前一个例子中，known 是在函数之外创建的，所以它属于被称为__main__ 的特殊帧。__main__ 之中的变量有时被称为全局 变量，因为它们可以在任意函数中访问。和局部变量在函数结束时就消失不同，全局变量可以在不同函数的调用之间持久存在。

全局变量常常用作标志 （flag）；它是一种布尔变量，可以标志一个条件是否为真。例如，有的函数使用一个叫verbose 的标志来控制输出的详细程度：

```
verbose = True

def example1():
    if verbose:
        print('Running example1')
```

如果你尝试给全局变量重新赋值，可能会感到惊讶。下面例子的本意是想记录函数是否被调用过：

```
been_called = False

def example2():
    been_called = True       # 错
```

但当你运行它时，会发现been_called 的值并不会变化。问题在于函数example2 会新建一个局部变量been_called 。局部变量在函数结束时就会消失，并且对全局变量没有任何影响。

要想在函数中给全局变量重新赋值，你需要在使用它之前先声明这个全局变量：

```
been_called = False

def example2():
    global been_called
    been_called = True
```

global 语句告诉编译器，“在这个函数里，当我说been_called 时，我指的是全局变量；不要新建一个局部变量。”

下面是一个尝试更新全局变量的例子：

```
count = 0

def example3():
    count = count + 1          # 错
```

如果运行它，会得到：

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python会假设count 是局部的，在这种假设下你在写入它之前先读取了。解决方案也是声明count 为全局变量。

```
def example3():
    global count
    count += 1
```

如果全局变量指向的是可变的值，可以不用声明该变量就可以修改该值：

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

所以你可以添加、删除和替换一个全局的列表或字典的元素，但如果想要给全局变量重新赋值，则需要声明它：

```
def example5():
    global known
    known = dict()
```

全局变量很有用，但是如果使用太多，并且频繁修改，可能会让代码比较难调试。

11.8　调试

在使用更大的数据集时，通过打印和手动检查输出的方式来调试已经变得很笨拙了。下面是一些调试大数据集的建议。

缩小输入

如果可能，减小数据集的尺寸。例如，程序如果读入文本文件，可以从开头10行开始，或者使用你能找到的最小样本。你可以编辑文件本身，或者（更好地）修改程序让它只读入前n 行。

如果出现了错误，可以调小n ，小到足够展现出错误的最小程度，并在修改之后逐渐增大n 。

检查概要信息和类型

与其打印和检查整个数据集，可以考虑打印出数据的概要信息：例如，字典中条目的数量，或者一个列表中数的和。

运行时错误的一个常见原因是某个值的类型不对。调试这种错误时，常常只需要打印出值的类型就足够了。

编写自检查逻辑

有时候可以写代码自动检查错误。例如，如果你要计算一系列数的平均值，可以检查结果是否比列表中最大的数小，或者比最小的数大。这种检查称为“健全检查”（sanity check），因为它会发现那些“发疯”的结果。

另一种检查可以对比两种不同的计算的结果，查看它们是否一致。这样的检查称为“一致性检查”。

格式化输出

格式化调试输出，可以更容易发现错误。我们在6.9节中已经看到过一个例子。`pprint` 模块提供了一个`pprint` 函数，可以将内置类型的值以更加人性化的可读的格式打印出来。（`pprint` 代表“pretty print”。）

另外，再提醒一次，花费时间构建脚手架代码，可以减少未来进行调试的时间。

11.9　术语表

映射（mapping）：一个集合中的每个元素与另一个集合中的元素所产生的关联。

字典 （dictionary）：从键到对应的值的映射。

键值对 （key-value pair）：键到值的映射的展示。

项 （item）：在字典中，键值对的另一个名称。

键 （key）：字典中出现在键值对的前一部分的对象。

值 （value）：字典中出现在键值对的后一部分的对象。这比我们

之前提到的“值”更加具体。

实现 （implementation）：进行计算的一个具体方式。

散列表 （hashtable）：Python字典的实现用的算法。

散列函数 （hash function）：散列表中用来计算一个键的位置的函数。

可散列 （hashable）：拥有散列函数的类型。不可变类型，诸如整数、浮点数和字符串都是可散列的；可变类型，诸如列表和字典，都是不可散列的。

查找 （lookup）：字典的一个操作，接收一个键，并找到它对应的值。

反向查找 （reverse lookup）：字典的一个操作，通过一个值来找到它对应的一个或多个键。

`raise` 语句 （raise statement）：一个（故意）抛出异常的语句。

单件 （singleton）：只包含一个元素的列表（或其他序列）。

调用图 （call graph）：一个用来展示程序运行中创建的每一帧的关系的图。使用箭头连接每个调用者和被调用者。

备忘 （memo）：将计算的结果存储起来，以避免将来进行不必要的计算。

全局变量 （global variable）：在函数之外定义的变量。全局变量可以在任何函数中访问。

全局语句 （global statement）：声明变量名为全局的语句。

标志 （flag）：用于标志一个条件是否为真的布尔变量。

声明 （declaration）：类似于`global`这样的用于通知解释器关于一个变量的信息的语句。

11.10　练习

练习**11-1**

编写一个函数，读入`words.txt`中 的单词，并将其作为键保存到一个字典中。字典的值是什么并不重要。然后你就可以使用`in`操作符快速检查一个字符串是否在这个字典中。

如果你做过了练习10-10，可以将这个实现与列表的`in`操作符以及二分查找进行速度的对比。

练习**11-2**

阅读字典方法`setdefault`的文档，并使用它来写一个更简洁的`invert_dict`。

解答：http://thinkpython2.com/code/invert_dict.py。

练习**11-3**

将练习6-2中的Ackermann函数改为备忘化的版本，并查看备忘化之后是否能让它运行更大的参数。提示：不能。

解答：http://thinkpython2.com/code/ackermann_memo.py。

练习**11-4**

如果你做过练习10-7，则已经有一个接受了列表作为形参的函数`has_duplicates`，当列表中有任意元素出现多于1次时返回`True`。

使用字典编写一个更快、更简单的`has_duplicates`。

解答：http://thinkpython2.com/code/has_duplicates.py。

练习**11-5**

两个单词，如果可以使用轮转操作将一个转换为另一个，则称为“轮转对”（参见练习 8-5中的`rotate_word`函数）。

编写一个程序，读入一个单词表，并找到所有的轮转对。

解答：http://thinkpython2.com/code/rotate_pairs.py。

练习**11-6**

下面是《车迷天下》节目中的另一个谜题（http://www.cartalk.com/content/ puzzlers）：

这个谜题是一个叫Dan O’Leary的伙计寄过来的。他曾经遇到一个单音节、5字母的常用单词，有如下所述的特殊属性。当你删除第一个

字母时，剩下的字母组成原单词的一个同音词，即发音完全相同的词。将第一个字母放回去，并删除第二个字母，结果也是原单词另一个同音词。问题是，这个单词是什么？

接下来我给你一个示例，但它并不能完全符合条件。我们看这个5字母单词 “wrack ”， W-R-A-C-K， 也就是“wrack with pain”（ “带来伤害”）里的那个词。如果我删掉第一个字母，会剩下一个4字母的单词，“R-A-C-K ”。也就是，“Holy cow, did you see the rack on that buck! It must have a nine-pointer!”（“天哪！ 你看到那匹雄鹿的鹿角了吗！ 一定有9个犄角！ ”）中的那个词。它是一个完美的同音词。但如果你把“w”放回去，并删掉“r”，会得到单词“wack”，也是一个真实单词，但它读音和其他两个不一样。

但就Dan和我所知，至少有一个单词能够通过删除前两个字母得到两个同音词。问题是，这个单词是什么？

你可以使用练习11-1中的字典来检测一个字符串是否出现在单词表中。

要检查两个单词是不是同音词，可以使用CMU发音词典。你可以从http://www.speech.cs.cmu.edu/ cgi-bin/cmudict 或者http://thinkpython2.com/ code/c06d下载它，也可以下载http://thinkpython2.com/code/pronounce.py，其中提供了一个叫作`read_dictionary`的函数来读入发音词典并返回一个Python字典，将每个单词映射到表示其主要读音的字符串上。

编写一个程序，列出所有可以解答这个谜题的单词。

解答：http://thinkpython2.com/code/homophone.py。

第12章　元组

本章介绍另外一种内置类型——元组，并展示列表、字典和元组三者如何一起工作。我还会介绍一种很有用的可变长参数列表功能：收集操作符和分散操作符。

请注意：元组（tuple）这个词的读音并没有统一标准。有些人会读成“tuh-ple”，与“supple”同音，但在程序设计界，大多数人都读作“too-ple”，与“quadruple”同音。

12.1　元组是不可变的

元组是值的一个序列。其中的值可以是任何类型，并且按照整数下标索引，所以从这方面看，元组和列表很像。元组和列表之间的重要区别是，元组是不可变的。

语法上，元组就是用逗号分隔的一列值：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

虽然并不必需，但元组常常用括号括起来：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

若要新建只包含一个元素的元组，需要在后面添加一个逗号：

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

而用括号括起来的单独的值并不是元组：

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

新建元组的另一种形式是使用内置函数`tuple`。不带参数时，它会新建一个空元组：

```
>>> t = tuple()
>>> t
()
```

如果参数是一个序列（字符串、列表或者元组），结果就是一个包含序列的元素的元组：

```
>>> t = tuple('lupins')
>>> t
```

```
('l', 'u', 'p', 'i', 'n', 's')
```

因为`tuple`是内置函数的名称，所以应当避免用它作为变量名称。

大多数列表操作也可以用于元组。方括号操作符可以用下标取得元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

而切片操作符选择一个范围内的元素：

```
>>> t[1:3]
('b', 'c')
```

但如果尝试修改元组中的一个元素，会得到错误：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

由于元组是不可变的，所以不能修改它的元素。但是可以将一个元

组替换为另一个：

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

这条语句生成新元组，然后使t 引用它。

关系运算符适用于元组和其他序列。Python从比较每个序列的第一个元素开始。如果它们相等，它就继续比较下一个元素，依次类推，直到它找到不同元素为止。子序列元素不在考虑之列（尽管它们实际上很大）。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2　元组赋值

交换两个变量的值常常很有用。使用传统的赋值方式，需要使用一个临时变量。例如，要交换a 和b ：

```
>>> temp = a
>>> a = b
>>> b = temp
```

```
```

这种解决方案很笨拙，而元组赋值 则更优雅：

```
>>> a, b = b, a
```

左边是一个变量的元组，右边是表达式的元组。每个值会被赋值给相应的变量。右边所有的表达式，都会在任何赋值操作进行之前完成求值。

左边变量的个数和右边值的个数必须相同：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

更通用地，右边可以是任意类型的序列（字符串、列表或元组）。例如，想要将电子邮件地址拆分成用户名和域名，可以这么写：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

split 返回两个元素的列表；第一个元素被赋值到uname ，第二个到domain 上。

```
>>> uname
'monty'
>>> domain
'python.org'
```

12.3　作为返回值的元组

严格地说，函数只能返回一个值，但如果返回值是元组的话，效果和返回多个值差不多。例如，如果将两个整数相除，得到商和余数，那么先计算x/y 再计算x%y 并不高效。更好的方法是同时计算它们。

内置函数divmod 接收两个参数，并返回两个值的元组，即商和余数。可以将结果存为一个元组：

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

或者可以使用元组赋值来分别存储结果中的元素：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

下面是返回一个元组的函数的示例：

```
def min_max(t):
    return min(t), max(t)
```

max 和min 都是内置函数，分别返回一个序列的最大值和最小值。min_max 计算这两个值并将它们作为一个元组返回。

12.4 可变长参数元组

函数可以接收不定个数的参数。以* 开头的参数名会收集（gather）所有的参数到一个元组上。例如，printall 接收任意个数的参数并打印它们：

```
def printall(*args):
    print (args)
```

收集参数可以使用任何你想要的名称，但按惯例通常使用args 。下面是函数如何工作的一个例子：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

收集的反面是分散（scatter）。如果有一个序列的值而想将它们作为可变长参数传入到函数中，可以使用* 操作符。例如，divmod 正好接收两个参数，但它不接收元组：

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

但如果将元组分散，就可以用了：

```
>>> divmod(*t)
(2, 1)
```

很多内置函数使用可变长参数元组。例如，max 和min 都可以接收任意个数的参数：

```
>>> max(1, 2, 3)
3
```

但是sum 并不这样。

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

作为练习，编写一个函数`sumall`，接收任意个数的参数并返回它们的和。

12.5　列表和元组

`zip` 是一个内置函数，接收两个或多个序列，并返回一个元组列表。每个元组包含来自每个序列中的一个元素。这个函数的名字取自拉链（zipper），它可以将两行链牙交替连接起来。

下面的例子将字符串和一个列表“拉”到一起：

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

结果是一个 **`zip`** 对象，它知道如何遍历每个元素对。使用`zip` 最常用的方式是在`for` 循环中：

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

zip 对象是一种迭代器，即用来迭代访问一个序列的对象。迭代器与列表有些方面类似，但与列表不同的是，迭代器不能使用下标来选择对象。

如果需要使用列表的操作符和方法，可以利用zip 对象制作一个列表：

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

结果是一个由元组组成的列表。在本例中，每个元组包含字符串中的一个字符，以及列表中对应的一个元素。

如果序列之间的长度不同，则结果的长度是所有序列中最短的那个：

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

可以在for 循环中使用元组赋值来访问元组的列表：

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print (number, letter)
```

每次循环中，Python选择列表中的下一个元组，并将其元素赋值给letter 和number 变量。这个循环的输出如下：

```
0 a
1 b
2 c
```

如果组合使用zip 、for 循环以及元组赋值，可以得到一种有用的模式，用于同时遍历两个或更多序列。例如，has_match 函数接收两个序列，t1 和t2 ，并当存在一个下标i 保证t1[i] == t2[i] 时返回True ：

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

如果需要遍历序列中的元素以及它们的下标，可以使用内置函数enumerate ：

```
for index, element in enumerate('abc'):
    print(index, element)
```

这个枚举的结果是一个枚举对象，这个对象迭代一个对序列，在这

个例子中，每个对都包含一个下标（从0开始）和一个来自给定序列的元素，输出结果还是：

```
0 a
1 b
2 c
```

12.6 字典和元组

字典有一个`items`方法可以返回一个元组的序列，其中每个元组是一个键值对：

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

结果是一个`dict_item`对象，它是一个迭代器，可以迭代访问每一个键值对。可以使用`for`循环来访问：

```
>>> for key, value in d.items()
...     print(key, value)
...
c 2
a 0
b 1
```

和预料中一样，字典中的项是没有特定顺序的。

从反方向出发，可以使用一个元组列表来初始化一个新的字典：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

组合使用`dict`和`zip`可以得到一个简洁的创建字典的方法：

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

字典方法`update`也接收一个元组列表，并将它们作为键值对添加到一个已有的字典中。

使用元组作为字典的键很常见（主要是因为不能使用列表）。例如，一个电话号码簿可能需要将姓名对映射到电话号码。假设定义了`last`，`first`和`number`，可以这么写：

```
directory[last,first] = number
```

在方括号中的表达式是一个元组。我们也可以使用元组赋值来遍历这个字典：

```
for last, first in directory:
    print(first, last, directory[last,first])
```

这个循环遍历字典directory 的所有键，它们都是元组。它将每一个元组的元素赋值给last 和first ，接着打印出名字以及对应的电话号码。

在状态图中有两种方法可以表达元组。更详细的版本和列表一样，显示索引和元素。例如，元组('Cleese', 'John') 可以如图12-1所示。

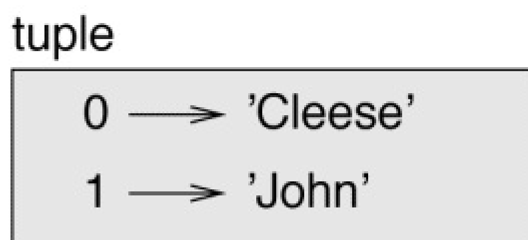


图12-1　状态图

但是在更大的图中你可能希望省略掉细节。例如，整个电话簿的图如图12-2所示。

dict

('Cleese', 'John') → '08700 100 222'
('Chapman', 'Graham') → '08700 100 222'
('Idle', 'Eric') → '08700 100 222'
('Gilliam', 'Terry') → '08700 100 222'
('Jones', 'Terry') → '08700 100 222'
('Palin', 'Michael') → '08700 100 222'

图12-2　状态图

这里元组使用Python的语法作为图形化的简写展示。这张图里的电话号码是BBC的投诉热线，所以请不要真去拨打它。

12.7　序列的序列

我一直在聚焦于元组的列表，但本章中几乎所有的示例都可以对列表的列表、元组的元组以及列表的元组使用。为了避免枚举所有的可能组合，有时候直接说序列的序列更简单。

在很多环境中，不同类型的序列（字符串、列表和元组）都可以互换使用。应当如何选择使用哪个呢？

从最明显的一个开始，字符串比其他序列有更多限制，因为它的元素必须是字符。它们也是不可变的。如果你需要修改一个字符串中的字符（而不是新建一个字符串），可能需要使用字符的列表。

列表比元组更加通用，主要因为它是可变的。但也有一些情况下你可能会优先选择元组。

1．在有些环境中，如返回语句中，创建元组比创建列表从语法上说更容易。

2．如果需要用序列作为字典的键，则必须使用不可变类型，如元组或字符串。

3．如果你要向函数传入一个序列作为参数，使用元组可能会减少潜在的由假名导致的不可预知行为。

因为元组是不可变的，它们不提供类似`sort`和`reverse`之类的方法，这些方法修改现有的序列。但Python也提供了内置函数`sorted`，可以接收任何序列作为参数，并按排好的顺序返回带有同样元素的新列表。Python还提供了`reverse`，可以接收序列作为参数，并返回一个以相反顺序遍历列表的迭代器。

12.8　调试

列表、字典和元组都被统一看作是一种数据结构 。本章中我们开始看到复合数据结构，像元组的列表，或者用元组做键、用列表做值的字典等。复合数据结构很有用，但它容易导致我称为的结构错误；也就是说，数据结构因为错的类型、大小或结构导致的错误。例如，如果你期望得到一个包含单个整数的列表，而我给你一个单个整数（而不是在列表中），就会出错。

为了帮助调试这种问题，我写了一个模块`structshape`，提供一个也叫作`structshape`的函数，接收任何数据类型作为参数，并返回一个描述它的形状的字符串。你可以从http://thinkpython2.com/code/structshape.py下载它。

下面是一个简单列表的结果：

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> structshape(t)
'list of 3 int'
```

更好看的程序可能会输出“list of 3 ints”，但不需要处理复数更加容易。下面是列表的列表：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

如果列表的元素不是同一种类型，`structshape`会根据它们的类型按顺序分组：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

下面是元组的列表：

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

下面是一个字典，有3个从整数映射到字符串的项：

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

如果你发现要记住数据结构有困难，`structshape` 可以帮忙。

12.9 术语表

元组 （tuple）：一个不可变的元素序列。

元组赋值 （tuple assignment）：一个赋值语句，右侧是一个序列，左侧是一个变量的元组。右边的序列会被求值，它的元素依次赋值给左侧元组中的变量。

收集 （gather）：组装可变长参数元组的操作。

分散 （scatter）：把一个序列当作参数列表的操作。

`zip` 对象 （zip object）：调用内置函数`zip` 的结果，它是一个迭代访问由元组组成的序列的对象。

迭代器 （iterator）：可以遍历序列的对象，但它不提供列表的操作和方法。

数据结构 （data structure）：相关的值的集合，通常组织成列表、字典、元组等。

结构错误 （shape error）：某个值由于其结构不对导致的错误，即它的类型或尺寸不对。

12.10　练习

练习**12-1**

编写一个函数`most_frequent` ，接收一个字符串并按照频率的降序打印字母。从不同语言中查找文本样例并查看不同语言中的单词频率如何变化。将你的结果和http://en.wikipedia.org/wiki/Letter_frequencies上的列表进行对比。

解答：http://thinkpython2.com/code/most_frequent.py。

练习**12-2**

更多回文！

1．编写一个程序从文件中读入一个单词列表（参见9.1节）并打印

出所有是回文的单词集合。

下面是输出的样子的示例：

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

提示：你可能需要构建一个字典将字母的集合映射到可以用这些字母构成的单词的列表上。问题是，如何表达字母集合，才能让它可以用作字典的键？

2．修改前一个问题的程序，让它先打印最大的回文列表，再打印第二大的回文列表，依次类推。

3．在Scrabble拼字游戏中，一个“bingo”代表你自己架子上全部7个字母和盘上的一个字母组合成一个8字母单词。哪一个8字母单词可以生成最多的bingo？提示：一共有7个。

解答：http://thinkpython2.com/code/anagram_sets.py。

练习**12-3**

两个单词，如果可以通过交换两个字母将一个单词转换为另一个，就称为“置换对”；例如，“converse”和“conserve”。编写一个程序查找字典中所有的置换对。提示：不要测试所有的单词对，也不要测试所有可能的交换。

解答：http://thinkpython2.com/code/metathesis.py。

鸣谢：这个练习启发自http://puzzlers.org的示例。

练习**12-4**

下面是《车迷天下》节目中的一个谜题（http://www.cartalk.com/content/puzzlers ）：

一个英文单词，当逐个删除它的字母时，仍然是英文单词。这样的单词中最长的是什么？

首先，字母可以从两头或者中间删除，但你不能重排字母。每次你去掉一个字母，则得到另一个英文单词。如果一直这么做，最终会得到一个字母，它本身也是一个英文单词——可以从字典上找到的。我想知道这样的最长的单词是什么，它有多少字母？

我会给你一个普通的例子：Sprite。你从sprite开始，取出一个字母，从单词内部取，取走r，这样我们就剩下单词spite，接着我们取走结尾的e，剩下spit，接着取走s，我们剩下pit、it和I。

编写一个程序来找到所有可以这样缩减的单词，然后找到最长的一个。

这个练习比大部分练习都更有挑战，所以下面有一些建议。

1．你可能需要编写一个程序接收一个单词，并计算出所有通过从它取出一个字母得到的单词的列表。它们是这个单词的“子”单词。

2．递归地，只有当一个单词的子单词中有一个可缩减时，它才可缩减。作为一个基准情形，你可以认为空字符串可缩减。

3．我提供的单词表，`words.txt`，并不存在单个字母的单词。所以你可能需要加上“I”、“a”和空字符串。

4．为了提高程序的效率，你可能需要记住已知的可缩减的单词。

解答：http://thinkpython2.com/code/reducible.py。

第13章　案例研究：选择数据结构

到这里你应该已经学会了Python的核心数据结构，也见过了一些使用它们的算法。如果你想要更多地了解算法，可以阅读第21章。但继续下面的内容之前那部分内容并不是必需要读懂，你可以随感兴趣时时去阅读。

本章配合练习介绍一个案例分析，帮你思考如何选择数据结构并如何实际使用它们。

13.1　单词频率分析

和前面的章节一样，应当至少尝试一下解决问题，再看我的解答。

练习**13-1**

编写一个程序，读入一个文件，将每行内容拆解为单词，剥去单词周围的空白字符和标点，并转换为小写。

提示：`string` 模块提供了空白字符串`whitespace`，包括空格、制表符、换行符等；它也提供了`punctuation`，包含了所有的标点字符。让我们试试能不能让Python胡言乱语：

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

另外，也可以考虑字符串方法`strip`、`replace`和`translate`。

练习**13-2**

去往古腾堡工程（Project Gutenberg，http://www.gutenberg.org）并下载你最喜欢的无版权书籍的纯文本文档。

修改前一个练习中的程序，改为从你下载的书籍中读取内容，跳过文件开头的信息部分，并和前面一样将文本处理成为单词。

接着修改程序，计算书中出现的全部单词的总数，以及每个单词使用的次数。

打印书中使用的不同单词的个数。比较不同时代、不同作者的不同书籍。哪一个作者使用的词汇最广泛？

练习**13-3**

修改前一个练习中的程序，计算书中使用频率最高的20个单词。

练习**13-4**

修改前面的程序，读入一个单词表（参见9.1节）并打印出书中所有不在单词表之中的单词。这其中有多少是拼写错误？有多少是应该出现在单词表中的常用单词？有多少是真正冷僻的单词？

13.2　随机数

给定相同的输入，大部分计算机程序每次运行都会生成相同的输出，所以它们被认为是有确定性 的。确定性通常是件好事，因为我们希望相同的计算能有相同的结果。但对某些特别的应用，我们希望计算机是不可预测的。游戏是一个明显的例子，但还有更多类似的例子。

让程序变得真正地不确定很难，但也有办法让它至少看起来是不确定的。一种办法是使用算法来生成伪随机数 。伪随机数并不是真正随机的，因为它们是通过一个确定性的算法生成的，但若只看输出的数字的话，几乎不可能看出来和随机数有什么区别。

模块random 提供了用于生成伪随机数的函数（接下来我直接简单地将它称为“随机数”）。

函数random 返回一个从0.0到1.0之间的随机浮点数（包括0.0，但不包括1.0）。每当调用random 时，会得到一个很长的随机数序列中的下一个数。运行下面的循环，可以看到一个样本：

```
import random

for i in range(10):
     x = random.random()
     print(x)
```

函数randint 接收参数low 和high ，并返回low 和high 之间（两者都包含）的一个整数。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

要从序列中随机选择一个元素，可以使用choice：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

random模块还提供了可以从各种连续分布序列中生成随机数的函数，包括高斯分布、指数分布、γ分布以及其他几种。

练习**13-5**

编写一个函数choose_from_hist，接收一个11.1节所定义的直方图作为参数，并从这个直方图中，按照频率的大小，成比例地随机返回一个值。例如，对下面这个直方图：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

你的函数应该以2/3的概率返回`'a'`，以1/3的概率返回`'b'`。

13.3　单词直方图

在继续阅读之前你应当尝试前面的练习。你可以从 http://thinkpython2.com/code/ analyze_book1.py 下载我的解答。你还需要 http://thinkpython2.com/code/emma.txt。

下面是一个读取文件并从文件中的单词构造直方图的例子：

```
import string

def process_file(filename):
     hist = dict()
     fp = open(filename)
     for line in fp:
          process_line(line, hist)
     return hist

def process_line(line, hist):
     line = line.replace('-', ' ')

     for word in line.split():
          word = word.strip(string.punctuation + string.whitespace)
          word = word.lower()

          hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

这个程序读入`emma.txt`，其内容是简·奥斯丁的《爱玛》的文本。

`process_file` 循环遍历文件中的每一行，每次将一行传递给

process_line 函数。直方图hist 用作累加器。

process_line 使用字符串方法replace 将'-' 符号替换为空格，再使用split 将各行文本拆分成一个字符串列表。它遍历单词列表，使用strip 和lower 去除掉标点符号并转换为小写。（我们说“转换”，只是个简称，记住字符串是不可变的，所以strip 和lower 这样的方法返回的是新字符串。）

最后，process_line 通过创建新项或者增加旧有项的值来更新直方图。

要计算文件中单词的总数，我们可以累加直方图中的频率：

```
def total_words(hist):
    return sum(hist.values())
```

不同单词的个数，就是字典里的元素数量：

```
def different_words(hist):
    return len(hist)
```

下面是打印结果的代码：

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

```
```

以及结果：

```
Total number of words: 161080
Number of different words: 7214
```

13.4　最常用的单词

要寻找最常用单词，我们可以生成一个元组的列表，其中每个元组包括一个单词及其频率，并对其进行排序。

下面的函数接收一个直方图，并返回“单词-频率”元组的列表：

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

在每个元组中，频率先出现，所以结果列表按频率排序。下面的循环打印出最常用的10个单词：

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
```

```
    print(word, freq, sep='\t')
```

这里我使用关键字参数sep 通知print 去使用制表符作为分隔符，而不使用空格。于是第二列可以对其排列。下面是《爱玛》的结果：

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

这段代码可以用sort 函数的key 参数进行简化。如果你有兴趣，可以读一下相关的文章：http://wiki.python.org/moin/HowTo/Sorting。

13.5 可选形参

我们已经见过一些接收可选形参的内置函数和方法。用户也可以编写接收可选形参的自定义函数。例如，下面的函数打印一个直方图中最常见的单词：

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
```

```
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

第一个形参是必需的；第二个是可选的。形参num 的默认值 是 10。

如果只提供一个实参：

```
print_most_common(hist)
```

num 会获得默认值。如果提供两个实参：

```
print_most_common(hist, 20)
```

num 则会获得所提供的实参值。换句话说，可选实参值覆盖 默认形参值。

如果一个函数既有必需形参，也有可选形参，则所有的必需形参都必须在前面，后面跟着可选形参。

13.6　字典减法

寻找在书中出现却不在words.txt 单词表中的单词，这个问题可以

看作是集合减法；也就是说，我们想要找到出现在一个集合（书中的单词）而不在另一个集合（单词表中的单词）的所有单词。

subtract 函数接收两个字典d1 和d2，并返回一个新的字典，包含所有出现在d1 中且不出现在d2 中的键值。由于我们并不真的关心字典的值，我们将所有值都设为None。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

要找出书中出现而不在words.txt 单词表中的词，我们可以使用process_file 为words.txt 建立一个直方图，再使用减法：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

下面是《爱玛》一书中的部分结果：

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

这些词中有些是名字或所有格单词。其他的，如“rencontre”，已经不再常用。但也有一些是真应该包含在单词表中的！

练习**13-6**

Python提供了一个数据结构`set`，它提供了很多常见的集合操作。你可以读19.5节中关于集合操作的内容，或者阅读http://docs.python.org/3/library/stdtypes.html#types-set上的文档，并编写一个程序使用集合减法来寻找出现在书中但不出现在单词表中的单词。解答：http://thinkpython2.com/code/ analyze_book2.py。

13.7 随机单词

若要从直方图中随机选择一个单词，最简单的算法是根据计算得到的频率构建一个列表，其中每个单词根据词频有多个拷贝，并从中随机选择一个单词：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

表达式`[word] * freq`创建一个列表，里面有单词`word`的`freq`

个副本。extend 方法和append 类似，区别是接收的参数是一个序列。

这个算法可以使用，但效率并不高；每当选择一个随机单词时，它会重建列表，而这个列表和原书差不多长。一个明显的改进方法是只建立列表一次，再使用多次选择，但这么做列表仍然很大。

更好的替代方案如下。

1．使用keys 来获得书中所有的单词的列表。

2．构建一个列表，包含单词频率的累积和（参见练习10-2）。这个列表中的最后一项是书中单词的总数*n* 。

3．在1到*n* 之间随机选择一个数。使用二分查找（参见练习10-11）来找到随机数在累积和列表中应该出现的位置的下标。

4．使用这个下标，在单词表中找到相应的单词。

练习**13-7**

编写一个程序，使用这个算法来从书中选择一个随机的单词。

解答：http://thinkpython2.com/code/analyze_book3.py 。

13.8　马尔可夫分析

如果你从书中随机地获取单词，可以借此感受一下书中的词汇，但可能无法通过随机获取来得到一句话：

```
this the small regard harriet which knightley's it most things
```

一个随机单词的序列，很难组成有意义的话，因为相邻的词之间没有任何关联。例如，在一个真实的句子中，冠词“the”应当会后接一个形容词或名词，而不应是动词或副词。

测量这种类型的关联的方法之一是使用马尔可夫分析，它能够用于描述给定的单词序列中下一个可能出现的单词的概率。例如，歌曲《Eric，the Half a Bee》的开头是：

Half a bee, philosophically,

Must, ipso facto, half not be.

But half the bee has got to be

Vis a vis, its entity. D’you see?

But can a bee be said to be

Or not to be an entire bee

When half the bee is not a bee

Due to some ancient injury?

在这段文本中，短语“half the”总是后接着单词“bee”，但短语“the

bee”则可能后接“has”或“is”。

马尔可夫分析的结果是一个从每个前缀（如“half the”和“the bee”）到其所有可能后缀（如“has”和“is”）的映射。

给定这种映射后，你就可以用它来生成随机文本。从任意前缀开始，并从它的可能后缀中随机选择一个。接着，你可以将前缀的结尾和后缀组合起来，作为下一个前缀，并继续重复。

例如，如果你以前缀“Half a”开始，则接下来一个单词必定是“bee”，因为这个前缀在文本中只出现了一次。下一个前缀是“a bee”，所以下一个后缀可能是“philosophically”“be”或者“due”。

在这个例子中前缀的长度总是2，但其实你可以使用任意前缀长度来进行马尔可夫分析。

练习**13-8**

马尔可夫分析：

1．编写一个程序从文件中读入文本，并进行马尔可夫分析。结果应该是一个字典，将前缀映射到可能后缀的集合。集合可以是列表、元组或者字典；由你来做出合适的选择。你可以使用前缀长度2来测试程序，但编写程序时应当考虑可以方便地改为其他前缀长度。

2．在前面编写的程序中添加一个函数，基于马尔可夫分析的结果随机生成文本。下面是一个从《爱玛》中使用前缀长度2生成的例子：

He was very clever, be it sweetness or be angry, ashamed or only

amused, at such a stroke. She had never thought of Hannah till you were never meant for me?” “I cannot make speeches, Emma:” he soon cut it all himself.

对这个例子，我留下了每个单词后面的标点。结果几乎是语法正确的，但也不完全对。语义上，它看起来很像是有意义的，但也不完全是。

当增加前缀长度时，结果会怎么样？随机生成的文本会不会看来更有意义？

3．一旦你的程序可以正常运行后，可以考虑尝试一下混搭：如果对两本或更多本书进行组合，则生成的随机文本会以一种有趣的方式混合各书中的词汇和短语。

致谢：本案例分析基于Kernighan和Pike的*The Practice of Programming*（Addison-Wesley, 1999）一书中的一个示例。

你应当在继续阅读前尝试这个练习，接着可从http://thinkpython2.com/code/ markov.py下载我的解答。你也需要http://thinkpython2.com/code/emma.txt。

13.9 数据结构

使用马尔可夫分析生成随机文本很有趣，但这个练习还有一个要点：数据结构的选择。在前面的练习中，你需要选择：

- 如何表达前缀；

- 如何表达可能的后缀的集合；
- 如何表达每个前缀到可能后缀的集合的映射。

最后一个选择很简单，要从键映射到对应的值，字典是最自然的选择。

对前缀来说，最明显的选择是字符串、字符串列表或者字符串元组。对后缀来说，一种选择是列表，另一种是直方图（字典）。

你会如何选择？第一步需要思考每种数据结构需要实现的操作。对前缀而言，我们需要能够从前方删除一个单词，并在后方添加一个单词。例如，如果当前的前缀是“Half a”，而下一个单词是“bee”，则需要能够构造下一个前缀，“a bee”。

你的第一个选择可能是列表，因为列表添加和删除元素都很方便。但我们也需要使用前缀作为字典的键，所以列表被排除掉。对元组而言，虽然你不能附加或删除，但可以使用加法操作符来构建一个新的元组：

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

shift 接收一个单词的元组、prefix，以及一个字符串word，并构建一个新的元组，包含prefix 中除了第一个之外的元素，并把word 添加在最后。

对后缀集合而言，我们需要进行的操作包括添加一个新的后缀（或者增加一个已有后缀的频率），以及随机选择一个后缀。

添加一个新后缀，使用列表实现或者直方图实现效率上相同。从一个列表中随机选择元素很简单；从直方图中随机选择则更难一些（参见练习13-7）。

到此为止我们一直在讨论实现的简易性，但选择数据结构时，还有其他需要考虑的因素。一个是运行时间。有时候，我们可以从理论上预期一种数据结构比另一种更快；例如，我提到过`in`操作符，当元素数量很大时，在字典中使用比在列表中快。

但哪种实现会更快常常无法事先预知。一个办法是两种都实现，再比较哪个更快。这种方法称为基准比较（benchmarking）。比较实际的方案是先选择最容易实现的数据结构，然后看它是否对预期的程序而言足够快。如果已经足够，则不需要变动；否则，可以使用`profile`模块之类的工具，发现程序中哪些地方占用了最长的时间。

另一个考虑因素是存储空间。例如，使用直方图来保存后缀集合可能占用较少空间，因为不论一个单词在文本中出现多少次，你只需要保存一次。有的情况下，节省空间也可以让你的程序运行更快，而在极端的情形中，如果导致内存溢出，则程序无法正常运行。但对大多数程序来说，存储空间是次于运行速度的第二考虑因素。

最后一点：在这个讨论中，对于分析和生成两个过程，我暗示了我们应当使用相同的数据结构。但因为这是两个分开的阶段，所以也可以在分析阶段使用一种数据结构，再转换为另一种数据结构用于生成阶

段。如果新的数据结构在生成阶段节省的时间大于转换花费的时间，则总的来说是有利的。

13.10　调试

当在调试程序时，尤其是对付一个困难的bug时，可以尝试下面5点。

阅读

审阅你的代码，对自己读出来，并检查它是否和你想说的一致。

运行

做一些小修改并进行试验，或者运行不同的版本。通常如果在程序中正确的地方加上正确的输出，问题就会变得更加显而易见。但有时候你需要构建一个脚手架。

沉思

花些时间思考！可能是哪种类型的错误：语法的、运行时的还是语义的？从错误消息或程序输出中可以得到什么信息？哪种错误可能导致你看到的问题？在问题出现之前，你的最后一次修改是什么？

橡皮鸭调试

如果你向其他人解释遇到的问题，有时候能在说完问题之前就找到答案。通常你甚至不需要找人去诉说，而只需要对橡皮鸭诉说即可。这就是著名的橡皮鸭调试（rubber duck debugging）的来源。这可不是我编出来的，参见https://en wikipedia.org/wiki/ Rubber_duck_debugging。

回退

在某种情况下，最好的办法就是回退，撤销最近的修改，直到你的程序恢复到之前没有错误且能够理解的程度。然后可以开始重新构建。

新手程序员有时会卡在这些环节中的某一个上，却忘了还可以尝试其他的环节。每个环节都有其独自的失败模式。

例如，当问题是一个拼写错误时，阅读代码可以帮忙，但若问题是概念误解导致，就没有效果了。如果你不理解自己的程序，那么即使阅读100遍，也发现不了问题，因为错误是在你大脑中的。

运行一些试验代码可以起到很大帮助，尤其是那些短小而简单的测试程序。但如果你没有思考或阅读代码就运行试验代码，则可能会陷入我称之为“随机走动编程”的模式之中。即毫无目标地随机改变程序，直到程序正确运行为止。毫无疑问，随机走动编程可能要花费很长的时间。

你必需花一定的时间去思考。调试就像是一门实验科学。你应当至少有一个关于这个问题的假设。如果有两个以上的可能性，可以试着构思一个测试来排除其中一个。

但如果有太多错误，或者你要修正的代码太大太复杂，即使最好的调试技巧也会失败。有时候最好的选择是回退，简化程序，直到得到一个你能够理解并且正确运行的程序。

新手程序员往往不愿意后撤，他们无法忍受删除一行代码（即使那是错误的代码）。如果能让你感觉更好，可以将程序复制到另外一个文件再开始删减它。这样以后就可以一点一点地复制回来。

寻找一个困难的bug，需要阅读、运行、沉思，甚至有时候需要回退。如果你在这其中一个环节上卡住了，可以尝试其他的环节。

13.11　术语表

确定性 （deterministic）：程序的一种特性：给定相同的输入，每次运行都会执行相同的操作。

伪随机 （pseudorandom）：一序列数：看起来是随机的，但实际上是由带着确定性的程序生成的。

默认值 （default value）：可选形参声明时给定的值，如果函数调用时没有指定这个实参的值，则使用该默认值。

覆盖 （override）：使用实参值替换一个默认值。

基准测试 （benchmarking）：实现不同的备选方案，并使用各种可能输入的样本来测试它们，以达到选择使用哪种数据结构的目的。

橡皮鸭调试 （rubber duck debugging）：通过向类似橡皮鸭之类的

静物解释你的问题，进行调试的过程。虽然橡皮鸭不懂Python，但通过诉说和解释，可以帮助你解决问题。

13.12　练习

练习**13-9**

一个单词的“排名”是它在单词列表中按频率排序的位置：最常见的词排名第1，次常用的词排第2，等等。

齐普夫定律（Zipf’s law）描述了排名和自然语言中词频的关系（http://en.wikipedia.org/ wiki/Zipfs_law）。特别地，它预测了排名为r的单词的频率f：

$$f = cr^{-s}$$

这里s 和c 是依赖于语言和文本的参数。如果在表达式两侧都调用对数，则得到：

$$\log f = \log c - s \log r$$

所以如果以$\log r$ 为横轴给$\log f$ 绘图，则会得到斜率为$-s$ ，截距为$\log c$ 的直线。

编写一个程序，从文件中读入文本，计算单词词频，并按照词频的降序，每一行打印出一个单词，以及$\log f$ 和$\log r$ 。使用你喜欢的制图程序将结果以图表形式展现出来，并检查它是否为直线。你能估计s 的值吗？

解答：http://thinkpython2.com/code/zipf.py。要运行我的解答，你需要安装绘图模块`matplotlib`。如果安装过Anaconda，你就已经有了`matplotlib`，否则你可能需要安装它。

第14章　文件

本章介绍“持久”程序的概念，它们将数据存储到持久存储中。另外，我们还会看到不同种类的持久存储，如文件和数据库。

14.1　持久化

我们现在见过的程序都是瞬态的，因为它们会在短暂的时间里运行出一些输出，但当运行结束后，它们的数据会消失。如果再次运行程序，它会再次全新地开始。

也有些程序是持久化 的：它们会运行很长一段时间（或者一直运行）；它们会至少存储一部分数据到永久存储（例如，硬盘）中；而且如果它们被关闭重启后，会接着从上次离开的状态继续。

持久化程序的例子包括操作系统，它几乎运行在任何一台开启的电脑中，以及web服务器，它们通常持续运行，等待网络上连入的请求。

读写文本文件是程序维护数据最简单的方法之一。我们已经见过读取文本文件的程序；在本章中将会见到往文件写入的程序。

另一种办法是将程序的状态保存到数据库中。本章中我们会介绍一个简单的数据库，以及一个模块，`pickle`，用来简化程序数据的存储。

14.2 读和写

文本文件是存储在诸如硬盘、闪存或光盘的永久媒介上的字符串序列。我们已经在9.1节中见过如何打开和读取一个文件。

要写入一个文件，需要使用'w' 模式作为第二个实参来打开它：

```
>>> fout = open('output.txt', 'w')
```

如果文件已经存在，则使用写模式打开时会清除掉旧有数据并重新开始，所以请谨慎！ 如果文件不存在，则会新建一个。

open 函数返回一个文件对象，提供操作文件的方法。其中write 方法把数据写入到文件中。

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

返回值是写入的字符数目。文件对象会记录写到了哪里，所以如果你再次调用write ，它会在文件的结尾处添加新的数据。

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

```

```

当写入完毕时，应该关闭文件。

```
>>> fout.close()
```

如果不关闭文件，程序会在执行结束时将文件关闭。

14.3　格式操作符

write 的参数必须是字符串，所以若我们想要往文件中写入其他类型的值，必须将它们先转换为字符串。最容易的办法是使用str：

```
>>> x = 52
>>> fout.write(str(x))
```

另一个办法是使用格式操作符% 。当用于整数时，% 是求余操作符。但若第一个操作对象是字符串时，% 则是格式操作符。

% 的第一个操作对象是格式字符串 ，包括了一个或多个格式序列 ，由它们来指定第二个操作对象如何格式化。表达式的结果是一个字符串。

例如，格式序列'%d' 意味着第二个操作数应该被格式化为十进制

整数。

```
>>> camels = 42
>>> '%d' % camels
'42'
```

结果是字符串'42'，请不要将它和整数值42混淆。

格式序列可以出现在字符串的任意地方，所以可以在一个句子中嵌入变量值：

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

如果字符串中有多于一个格式序列，第二个操作对象就必须是元组。每个格式序列按顺序对应元组中的一个元素。

下面的例子使用'%d' 格式化整数，'%g' 格式化浮点数，以及'%s' 格式化字符串：

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

元组中元素的个数必须和字符串中格式序列的个数一致。另外，元

素的类型也要和格式序列一致：

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

第一个例子中，元组中元素个数不够；第二个例子中，元素的类型不对。

更多关于格式操作符的信息参见 https://docs.python.org/3/library/stdtypes.html#printf-style- string-formatting。还有一个更强大的替代方案是字符串格式方法，参见 https://docs.python.org/3/library/stdtypes.html#str.format。

14.4 文件名和路径

文件组织在目录（也称为文件夹）中。每个程序都有“当前目录”，它是大多数操作的默认目录。例如，当打开一个文件用于读取时，Python默认在当前目录寻找它。

os 模块提供了用于操作文件和目录的函数（os代表operating system，即操作系统）。os.getcwd 返回当前目录的名称：

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

cwd 表示current working directory（即“当前工作目录”）。这个例子里的结果是/home/dinsdale，是名为dinsdale 的用户的主目录。

类似于'/home/dinsdale' 这样用来定位一个文件或目录的字符串被称为一个路径 （path）。

而一个简单文件名，如memo.txt，也被认为是一个路径，但它是一个相对路径，因为它依赖于当前目录。如果当前目录是/home/dinsdale，则文件名memo.txt 指的是/home/dinsdale/memo.txt。

而以/开头的路径则不依赖于当前目录，所以被称为绝对路径（absolute path）。可以使用os.path.abspath 来找寻文件的绝对路径：

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path 还提供了其他函数来操作文件名和路径。例如，os.path.exists 检查一个文件或目录是否存在：

```
>>> os.path.exists('memo.txt')
True
```

如果它存在，os.path.isdir 可以检查它是否为目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

类似地，os.path.isfile 检查它是否为文件。

os.listdir 返回指定目录中文件（以及其他目录）的列表：

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

为了演示这些函数，下面的例子“走遍”一个目录，打印所有文件的名称，并对之中的子目录递归调用自己。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join 接收一个目录和一个文件名称，并将它们拼接为一个完整的路径。

os 模块提供了一个函数walk，和上面的例子作用类似，但功能更丰富。作为练习，请阅读文档，并使用它打印指定目录中文件的名称和它的子目录。你可以从http://thinkpython2.com/code/walk.py下载我的解答。

14.5 捕获异常

当尝试读取和写入文件时，很多东西都可能出错。如果尝试打开一个不存在的文件，会得到一个IOError：

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

如果没有权限访问一个文件：

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

如果尝试打开一个目录用于文件读取，会得到：

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

要避免这些错误，可以使用类似`os.path.exists` 和`os.path.isfile` 的函数，但要检查所有的可能需要花费大量时间和代码（“Errno 21”这个名字，说明至少有21种可能出错的地方）。

最好是直接去尝试——等发生问题时再去解决它们——这也正是`try` 语句所做的事情。语法和`if…else` 语句类似：

```
try:
    fin = open('bad_file')
except:
    print ('Something went wrong.')
```

Python会先从`try` 子句开始，如果一切顺利，则跳过`except` 语句并继续执行。如果发生了异常，则跳出`try` 子句，并运行`except` 子句。

使用`try` 语句处理异常的过程称为捕获 一个异常。在这个例子里，`except` 语句打印的错误信息并没有太多用处。总的来说，捕获异常给了你一个修补错误的机会，或者可以再次尝试，或者至少能够优雅地停止程序。

14.6　数据库

数据库 是一个有组织的用于存储数据的文件。许多数据库都像字

典一样组织数据，因为它们也将键映射到值上。数据库和字典之间最大的区别是数据库是保存在磁盘上（或者其他永久存储上）的，所以当程序结束时它也能持续存在。

模块dbm 提供了接口用于创建和更新数据库文件。作为示例，我将会创建一个数据库保存图片文件的标题。

打开一个数据库和打开其他类型的文件差不多：

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

模式'c' 意味着数据库应当被创建，如果它不存在的话。调用的结果是一个数据库对象，（对大多数操作）可以当作字典来用。

当创建一个新项时，dbm 会更新数据库文件。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

当访问数据库中的一项时，dbm 会读取文件：

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

这里的结果是一个字节组对象（bytes object），因此以b 开头。字节组对象和字符串很类似。当你更加深入研究Python的时候，它们的区别可能会变得很重要，但现在可以忽略。

如果对一个已经存在的键赋值，dbm 会替换旧值：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

有一些字典方法，如keys 和items ，对数据库对象不可以使用。但使用for 循环来迭代遍历是可以的：

```
for key in db:
    print(key, db[key])
```

和其他文件一样，当操作结束时，需要关闭数据库：

```
>>> db.close()
```

14.7　封存

dbm 的限制之一是键和值都必须是字符串或字节。如果尝试使用其

他类型，则会出现错误。

pickle 模块可以帮忙。它可以将几乎所有类型的对象转换为适合保存到数据库的字符串形式，并可以将字符串转换回来成为对象。

pickle.dumps 接收一个对象作为参数，并返回它的字符串表达形式（dumps 是“dump string”的简写，意即转储字符串）：

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

这个格式不适合人眼阅读；它是为了方便pickle 模块的转换而设计的。pickle.loads （load string，即加载字符串）重新构造对象：

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

虽然新的对象和旧有对象的值相同，但（通常来说）它们不是同一个对象：

```
>>> t1 == t2
True
>>> t1 is t2
False
```

也就是说，封存再解封，和复制对象效果相同。

你可以使用`pickle`向数据库存储非字符串的值。事实上，这个组合如此常用，以至于Python已经将它们封装起来成为一个模块，叫作`shelve`。

14.8 管道

大部分操作系统都提供了命令行接口，也称为字符界面（shell）。字符界面通常会提供命令来浏览文件系统和启动应用程序。例如，在Unix中，可以使用`cd`来更换目录，使用`ls`来展示目录中的内容，以及打入`firefox`来启动浏览器。

任何在字符界面能启动的程序都可以在Python中使用管道对象（pipe object）来启动。管道对象代表一个正在运行的程序。

例如，Unix命令`ls -l`以长格式展示当前目录的内容。可以使用`os.popen`[1]来启动`ls`：

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

参数是一个字符串，它包含一个shell命令。返回值是一个和打开的

文件差不多的对象。可以使用readline 来逐行读取ls 进程的输出，或者使用read 一次读取所有输出：

```
>>> res = fp.read()
```

当你完成时，可以像文件一样关闭这个管道：

```
>>> stat = fp.close()
>>> print(stat)
None
```

返回值是ls 进程的最终状态；None 代表它正常结束了（没有错误）。

例如，大部分Unix系统都提供了一个叫作md5sum 的命令，它读取文件的内容并计算出一个“校验和”（checksum）。你可以在http://en.wikipedia.org/wiki/Md5阅读MD5的相关信息。这个命令提供了一个高效的方法，用来对比两个文件是否包含相同的内容。不同的内容生成相同的校验和的概率极低（也就是，在宇宙崩溃之前不大可能发生）。

可以在Python中使用管道来运行md5sum ，并获得结果：

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
```

```
>>> res = fp.read()
>>> stat = fp.close()
>>> print res
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9 编写模块

任何包含Python代码的文件都可以作为模块导入。例如，如果你有一个文件`wc.py`，其代码如下：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

如果你运行这个程序，它会读取自身的内容，并打印出文件的行数，即7。你也可以像这样导入它：

```
>>> import wc
7
```

现在你有一个模块对象`wc`了：

```
>>> wc
<module 'wc' from 'wc.py'>
```

该模块对象提供了`linecount`：

```
>>> wc.linecount('wc.py')
7
```

上述就是在Python中编写模块的方法。

这个例子唯一的问题是当你导入模块时，它会运行底部的测试代码。正常情况下，当你导入一个模块时，它会定义新的函数，但不会运行。

作为模块导入的程序，通常使用如下模式：

```
if _ _name_ _ == '_ _main_ _':
    print(linecount('wc.py'))
```

`__name__` 是一个内置变量，当程序启动时就会被设置。如果程序作为脚本执行，`__name__` 的值是`'__main__'`；此时，测试代码会被运行。否则，如果程序作为模块被导入，则测试代码就被跳过了。

作为练习，把这个例子输入到一个文件`wc.py` 中，并将它作为一个

脚本运行。然后运行Python解释器，并导入wc 。当模块被导入时，__name__ 的值是什么？

警告：如果你导入一个已经被导入的模块，Python什么都不做。它不会重新读取文件，即使文件已经修改。

如果你想要重载一个模块，可以使用内置函数reload ，但它也可能会有棘手的问题。所以最安全的办法是重启解释器，并再次导入模块。

14.10 调试

当你读取和写入文件时，可能会遇到和空白字符相关的问题。这些问题可能会很难调试，因为空格、制表符和换行符通常都是不可见的：

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2	 3
 4
```

内置函数repr 可以帮忙。它接收任何对象作为参数，并返回对象的字符串表达形式。对于字符串来说，它使用反斜杠序列来展示空白字符：

```
>>> print (repr(s))
'1 2\t 3\n 4'
```

这样可以帮助调试。

另一个你可能遇到的问题是不同的系统使用不同的字符表示换行。有的系统使用一个换行符，即`\n`。另外的系统使用一个回车符，即`\r`。也有的系统两者都使用。如果你在不同的系统间移动文件，这些不一致之处可能会导致问题。

大多数系统都有程序可以将一种格式转换为另一种。你可以在 http://en.wikipedia.org/wiki/ Newline找到它们（并阅读这个问题的更多信息）。或者，当然，你也可以自己写一个。

14.11 术语表

持久性（persistent）：程序的一种属性，它会一直运行，并至少保存一部分数据在永久存储中。

格式操作符（format operator）：一个操作符，即`%`，它接收一个格式字符串，以及一个元组，并生成字符串，其中包括了元组的各个依据格式字符串里指定的方式格式化的元素。

格式字符串（format string）：一个字符串，被格式操作符所用，内部包含格式序列。

格式序列（format sequence）：格式字符串中出现的字符序列，如`%d`，它指定一个值如何格式化。

文本文件 （text file）：存储在类似硬盘这样的永久存储中的字符串序列。

目录 （directory）：有名称的文件集合。也称为文件夹。

路径 （path）：用来标定一个文件的字符串。

相对路径 （relative path）：从当前目录开始的路径。

绝对路径 （absolute path）：从文件系统的顶级目录开始的路径。

捕获 （catch）：使用`try`和`except`语句来阻止一个异常终止程序的行为。

数据库 （database）：一个文件，其内容组织类似于字典，将键映射到值。

字节组对象 （bytes object）：一个和字符串相似的对象。

命令行 （shell）：一个程序，允许用户键入命令并通过调用其他程序来执行这些命令。

管道对象 （pipe object）：代表一个运行中的程序的对象，让Python程序可以运行命令并读取结果。

14.12　练习

练习**14-1**

写一个名为sed的 函数，接收如下参数：一个模式字符串，一个替换用字符串，以及两个文件名。它应该读取第一个文件，并将内容写入第二个文件（如果需要则新建它）。如果文件中任何地方出现了模式字符串，应该替换掉。

如果在打开、读取、写入或关闭文件的过程中遇到错误，你的程序应当能捕获异常，打印一个错误信息，并退出。

解答：http://thinkpython2.com/code/sed.py。

练习**14-2**

如果你从http://thinkpython2.com/code/anagram_sets.py下载我对练习12-2的解答，你会发现它创建一个字典，将一个排好序的字母串映射到可以由这些字母组成的单词的列表。例如，'opst' 映射到 ['opts'，'post'，'pots'，'spot'，'stop'，'tops'] 列表。

编写一个模块，导入anagram_sets ，并提供两个新函数：store_anagrams 应当存储回文字典到一个“shelf”中；read_anagrams 应当查询一个单词，并返回它的回文的列表。

解答：http://thinkpython2.com/code/anagram_db.py。

练习**14-3**

在一个庞大的MP3文件的集合中，有可能同一首歌有多个副本，保存在不同的目录中，或者文件名不同。这个练习的目的是搜索重复的

歌。

1．编写一个程序递归搜索目录及其所有的子目录，并返回所有指定后缀（如`.mp3`）的文件的完整路径的列表。提示：`os.path`提供了几个有用的方法来操纵文件和路径名称。

2．要发现重复文件，需要使用`md5sum`来计算每个文件的“校验和”。如果两个文件的校验和相同，它们很可能有相同的内容。

3．你可以使用Unix命令`diff`来复审检验。

解答：http://thinkpython2.com/code/find_duplicates.py

---

[1] `popen`现在已经计划废止了，也就是说我们应当不再使用它，而是开始使用`subprocess`模块。但对于简单的情形，我发现`subprocess`过度复杂了。所以我仍然继续使用`popen`，直到它被完全废止。

第15章　类和对象

到现在你已经知道如何使用函数来组织代码，以及如何用内置类型来组织数据。下一步将学习“面向对象编程”，面向对象编程使用自定义的类型同时组织代码和数据。面向对象编程是一个很大的话题，需要好几章来讨论。

本章的代码示例可以从http://thinkpython2.com/code/Point1.py下载，练习的解答可以在http://thinkpython2.com/code/Point1_soln.py下载。

15.1　用户定义类型

我们已经使用了很多Python的内置类型；现在我们要定义一个新类型。作为示例，我们将会新建一个类型`Point`，用来表示二维空间中的一个点。

在数学的表示法中，点通常使用括号中逗号分隔两个坐标表示。例如，$(0, 0)$表示原点，而(x, y)表示一个在原点右侧x单位，上方y单位的点。

在Python中，有好几种方法可以表达点。

- 我们可以将两个坐标分别保存到变量`x`和`y`中。
- 我们可以将坐标作为列表或元组的元素存储。
- 我们可以新建一个类型用对象表达点。

新建一个类型比其他方法更复杂一些，但它的优点很快就会显现出来。

用户定义的类型也称为类（class）。类的定义如下所示：

```
class Point:
    """Represents a point in 2-D space."""
```

定义头表示新的类名为`Point`。定义体是一个文档字符串，解释这个类的用途。可以在类定义中定义变量和函数，我们会在后面回到这个话题。

定义一个叫作`Point`的类会创建一个对象类（object class）。

```
>>> Point
<class '__main__.Point'>
```

因为`Point`是在程序顶层定义的，它的“全名”是`__main__.Point`。

类对象像一个创建对象的工厂。要新建一个`Point`对象，可以把`Point`当作函数来调用：

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

返回值是到一个`Point`对象的引用，我们将它赋值给变量`blank`。

新建一个对象的过程称为实例化（instantiation），而对象是这个类的一个实例。

在打印一个实例时，Python会告诉你它所属的类型，以及存储在内存中的位置（前缀`0x`表示后面的数字是十六进制的）。

每个对象都是某个类的实例，所以“对象”和“实例”这两个词很多情况下都可以互换。但是，在本章中我使用“实例”来表示一个自定义类型的对象。

15.2 属性

可以使用句点表示法来给实例赋值：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

这个语法和从模块中选择变量的语法类似，如`math.pi`或者`string.whitespace`。但在这种情况下，我们是将值赋给一个对象的有命名的元素。这些元素称为属性（attribute）。

作为名词时，“AT-trib-ute”发音的重音在第一个音节，这与作为动

词的“a-TRIB-ute”不同。

下面的图表展示了这些赋值的结果。展示一个对象和其属性的状态图称为对象图 （object diagram），参见图15-1。

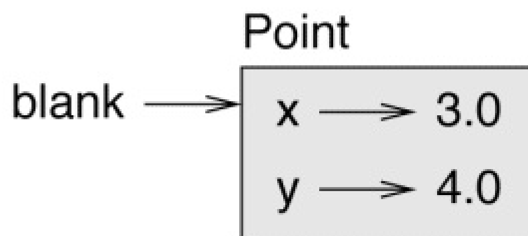


图15-1　对象图

变量blank 引用向一个Point对象，它包含了两个属性。每个属性引用一个浮点数。

可以使用相同的语法来读取一个属性的值：

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表达式blank.x 表示，“找到blank 引用的对象，并取得它的x 属性的值”。在这个例子中，我们将那个值赋值给一个变量x 。变量x 和属性x 并不冲突。

可以在任意表达式中使用句点表示法。例如：

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

可以将一个实例作为实参按通常的方式传递。例如：

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

print_point 接收一个点作为形参，并按照数学表达式展示它。可以传入blank 作为实参来调用它：

```
>>> print_point(blank)
(3.0, 4.0)
```

在函数中，p 是blank 的一个别名，所以如果函数修改了p，则blank 也会改变。

作为练习，编写一个叫作distance_between_points 的函数，接收两个Point对象作为形参，并返回它们之间的距离。

15.3　矩形

有时候对象应该有哪些属性非常明显，但也有时候需要你来做决定。例如，假设你在设计一个表达矩形的类。你会用什么属性来指定一个矩形的位置和尺寸呢？可以忽略角度，为了简单起见，假定矩形不是垂直的就是水平的。

最少有以下两种可能。

- 可以指定一个矩形的一个角落（或者中心点）、宽度以及高度。
- 可以指定两个相对的角落。

现在还很难说哪一种方案更好，所以作为示例，我们仅先实现第一个。

下面是这个类的定义：

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

文档字符串列出了属性：`width` 和`height` 是数字；`corner` 是一个Point对象，用来指定左下角的顶点。

要表达一个矩形，需要实例化一个Rectangle对象，并对其属性赋值：

```
box = Rectangle()
box.width = 100.0
```

```
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表达式box.corner.x 表示，“去往box 引用的对象，并选择属性corner ；接着去往那个对象，并选择属性x ”。

图15-2展示了这个对象的状态。作为另一个对象的属性存在的对象是内嵌 的。

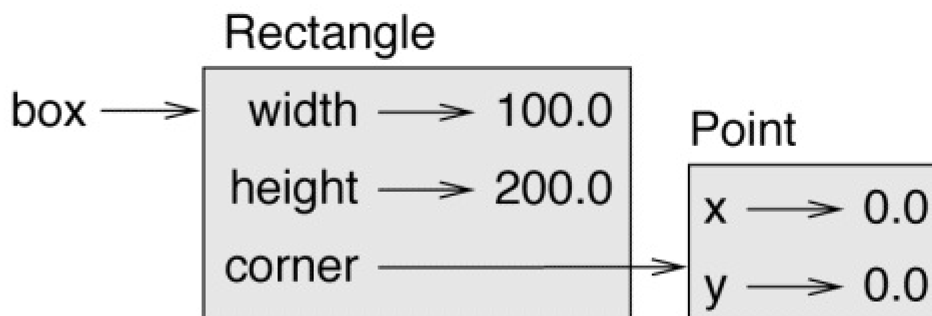


图15-2　对象图

15.4　作为返回值的实例

函数可以返回实例。例如，find_center 接收Rectangle 对象作为参数，并返回一个Point 对象，包含这个Rectangle 的中心点的坐标：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
```

```
    return p
```

下面是一个示例，传入box 作为实参，并将结果的Point对象赋给变量center：

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5　对象是可变的

可以通过给一个对象的某个属性赋值来修改它的状态。例如，要修改一个矩形的尺寸而保持它的位置不变，可以修改属性width 和height 的值：

```
box.width = box.width + 50
box.height = box.height + 100
```

也可以编写函数来修改对象。例如，grow_rectangle 接收一个Rectangle对象和两个数，dwidth 和dheight，并把这些数加到矩形的宽度和高度上：

```
def grow_rectangle(rect, dwidth, dheight):
```

```
    rect.width += dwidth
    rect.height += dheight
```

下面是展示这个函数效果的示例：

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

在函数中，rect 是box 的别名，所以如果当修改了rect 时，box 也改变。

作为练习，编写一个名为move_rectangle 的函数，接收一个Rectangle对象和两个分别名为dx 和dy的 数值。它应当通过将dx 添加到corner 的x 坐标和将dy 添加到corner 的y 坐标来改变矩形的位置。

15.6 复制

别名的使用有时候会让程序更难阅读，因为一个地方的修改可能会给其他地方带来意想不到的变化。要跟踪掌握所有引用到一个给定对象的变量非常困难。

使用别名的常用替代方案是复制对象。copy 模块里有一个函数copy 可以复制任何对象：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 和p2 包含相同的数据，但是它们不是同一个Point对象。

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

正如我们预料，is 操作符告诉我们p1 和p2 不是同一个对象。但你可能会预料== 能得到True 值，因为这两个点包含相同的数据。如果那样，你会失望地发现对于实例来说，== 操作符的默认行为和is 操作符相同，它会检查对象同一性，而不是对象相等性。这是因为对于用户自定义类型，Python并不知道怎么才算相等。至少现在还不行。

如果使用copy.copy 复制一个Rectangle，你会发现它复制了Rectangle对象但并不复制内嵌的Point对象：

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
```

```
>>> box2.corner is box.corner
True
```

图15-3展示了这个操作的对象图。这个操作称为浅复制 （shallow copy），因为它复制对象及其包含的任何引用，但不复制内嵌对象。

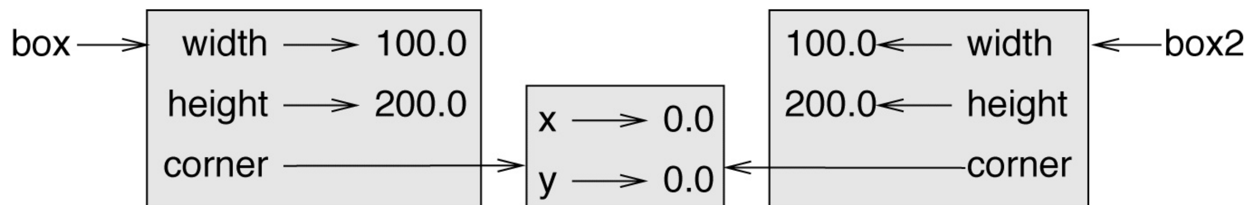


图15-3　对象图

对于大多数应用，这并不是你所想要的。在这个例子里，对一个Rectangle对象调用`grow_rectangle` 并不会影响其他对象，但对任何一个Rectangle对象调用`move_rectangle` 都会影响全部两个对象！这种行为既混乱不清，又容易导致错误。

幸好，`copy` 模块还提供了一个名为`deepcopy`的 方法，它不但复制对象，还会复制对象中引用的对象，甚至它们 引用的对象，依次类推。所以你并不会惊讶这个操作为何称为深复制 （deep copy）。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 和box 是两个完全分开的对象。

作为练习，编写move_rectangle 的另一个版本，它会新建并返回一个Rectangle对象，而不是直接修改旧对象。

15.7　调试

开始操作对象时，可能会遇到一些新的异常。如果试图访问一个并不存在的属性，会得到AttributeError：

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

如果不清楚一个对象是什么类型，可以问：

```
>>> type(p)
<class '_ _main_ _.Point'>
```

也可以使用isinstance 来检查对象是否是某个类的实例：

```
>>> isinstance(p,Point)
True
```

如果不确定一个对象是否拥有某个特定的属性，可以使用内置函数hasattr：

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一个形参可以是任何对象，第二个形参是一个包含属性名称的字符串 。

也可以使用try 语句来尝试对象是否拥有你需要的属性：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

这种方法可以使编写适用于不同类型的函数更加容易。关于这一主题的更多内容参见17.9节。

15.8　术语表

类 （class）： 一个用户定义的类型。类定义会新建一个类对象。

类对象（class object）：一个包含用户定义类型的信息的对象。类对象可以用来创建该类型的实例。

实例（instance）：属于某个类的一个对象。

实例化（instanciate）：创建一个新对象。

属性（attribute）：一个对象中关联的有命名的值。

内嵌对象（embedded object）：作为一个对象的属性存储的对象。

浅复制（shallow copy）：复制对象的内容，包括内嵌对象的引用；`copy` 模块中的`copy` 函数实现了这个功能。

深复制（deep copy）：复制对象的内容，也包括内嵌对象，以及它们内嵌的对象，依次类推；`copy` 模块中的`deepcopy` 函数实现了这个功能。

对象图（object diagram）：一个展示对象、对象的属性以及属性的值的图。

15.9　练习

练习**15-1**

定义一个新的名为`Circle` 的类表示圆形，它的属性有`center` 和`radius` ，其中`center` 是一个Point对象，而`radius` 是一个数。

实例化一个Circle对象来代表一个圆心在(150, 100)、半径为75的圆

形。

编写一个函数`point_in_circle`，接收一个Circle对象和一个Point对象，并当Point处于Circle的边界或其内时返回`True`。

编写一个函数`rect_in_circle`，接收一个Circle对象和一个Rectangle对象，并在Rectangle的任何一个角落在Circle之内时返回True。另外，还有一个更难的版本，需要在Rectangle的任何部分都落在圆圈之内时返回True。

解答：http://thinkpython2.com/code/Circle.py 。

练习**15-2**

编写一个名为`draw_rect`的函数，接收一个Turtle对象和一个Rectangle对象作为形参，并使用Turtle来绘制这个Rectangle。如何使用Turtle对象的示例参见第4章。

编写一个名为`draw_rect`的函数，接收一个Turtle对象和一个Circle对象，并绘制出Circle。

解答：http://thinkpython2.com/code/draw.py 。

第16章　类和函数

现在我们已经知道如何创建新的类型，下一步是编写接收用户定义对象作为参数或者将其当作结果返回的函数。本章我会展示“函数式编程风格”，以及两个新的程序开发计划。

本章的代码示例可以从http://thinkpython2.com/code/Time1.py下载。练习的解答在http://thinkpython2.com/code/Time1_soln.py。

16.1　时间

作为用户定义类型的另一个例子，我们定义一个叫作`Time`的类，用于记录一天里的时间。类定义如下：

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

我们可以创建一个`Time`对象并给其属性小时数、分钟数和秒钟数赋值：

```
time = Time()
time.hour = 11
time.minute = 59
```

```
time.second = 30
```

Time 对象的状态图参见图16-1。

作为练习，编写一个叫作print_time 的函数，接收一个Time对象作为形参并以“hour:minute:second ”的格式打印它。提示：格式序列'%.2d' 可以以最少两个字符打印一个整数，如果需要，它会在前面添加前缀0。

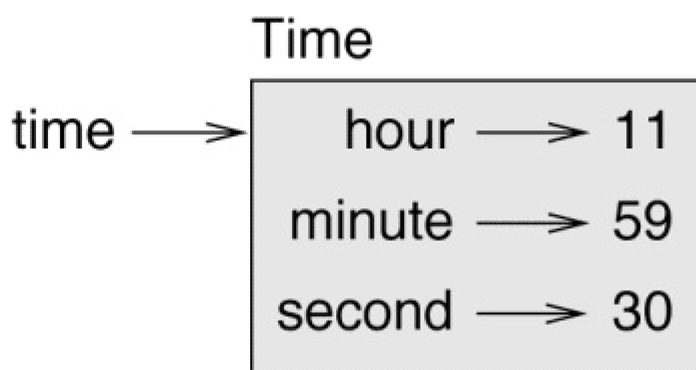


图16-1　对象图

编写一个布尔函数is_after ，接收两个Time对象，t1 和t2 ，并若t1 在t2 时间之后则返回True ，否则返回False 。挑战：不许使用if 表达式。

16.2　纯函数

在下面几节中，我们会编写两个用来增加时间值的函数。它们展示了两种不同类型的函数：纯函数和修改器。它们也展示了我会称为原型

和补丁 （prototype and patch）的开发计划。这是一种对应复杂问题的方法，从一个简单的原型开始，并逐渐解决更多的复杂情况。

下面是`add_time` 的一个简单原型：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

这个函数创建一个新的Time对象，初始化它的属性，并返回这个新对象的一个引用。这被称为一个纯函数 ，因为它除了返回一个值之外，并不修改作为实参传入的任何对象，也没有任何如显示值或获得用户输入之类的副作用。

为了测试这个函数，我将创建两个Time对象：`start` ，存放一个电影（如*Monty Python and the Holy Grail* ）的开始时间；`duration` ，存放电影的播放时间，在这里是1小时35分钟。

`add_time` 计算出电影何时结束。

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0
```

```
>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

结果`10:80:00`可能并不是你所期望的。问题在于这个函数并没有处理好秒数或者分钟数超过60的情况。当此发生时，我们需要将多余的秒数“进位”到分钟数，将多余的分钟数“进位”到小时数。

下面是一个改进的版本：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

虽然这个函数是正确的，它已经开始变大了。我们会在后面看到一个更短的版本。

16.3 修改器

有时候用函数修改传入的参数对象是很有用的。在这种情况下，修改对调用者是可见的。这样工作的函数称为修改器 （modifier）。

函数increment 给一个Time 对象增加指定的秒数，可以自然地写为一个修改器。下面是一个初稿：

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

第一行进行基础操作；后面的代码处理我们前面看到的特殊情况。

这个函数正确吗？如果seconds 比60大很多，会发生什么？

在那种情况下，只进位一次是不够的；我们需要重复进位，直到time.second 比60小。一个办法是使用while 语句替代if 语句。那样会让函数变正确，但并不很高效。作为练习，编写正确的increment 版本，并不包含任何循环。

任何可以使用修改器做到的功能都可以使用纯函数实现。事实上，有的编程语言只允许使用纯函数。有证据表明使用纯函数的程序比使用修改器的程序开发更快，错误更少。但有时候修改器还是很方便的，并且函数式程序的运行效率不那么高。

总的来说，我推荐你只要合理的时候，都尽量编写纯函数，而只在有绝对说服力的原因时才使用修改器。这种方法可以称作函数式编程风格 。

作为练习，编写一个`increment`的纯函数版本，创建并返回一个新的Time对象，而不是修改参数。

16.4　原型和计划

刚才我展示的开发计划称为“原型和补丁”。对每个函数，我编写一个可以进行基本计算的原型，再测试它，从中发现错误并打补丁。

这种方法在对问题的理解并不深入时尤其有效。但增量地修正可能会导致代码过度复杂（因为它们需要处理很多特殊情况），并且也不够可靠（因为很难知道你是否已经找到了所有错误）。

另一种方法是有规划开发 （designed development）。对问题有更高阶的理解能够让编程简单得多。在上面的问题中，如果更深入地理解，可以发现Time对象实际上是六十进制数里的3位数（参见http://en.wikipedia.org/wiki/Sexagesimal）！`second`属性是“个位数”，`minute`属性是“60位数”，而`hour`属性是“360位数”。

在编写`add_time`和`increment`时，我们实际上是在六十进制上进行加减，因此才需要从一位进位到另一位。

这个观察让我们可以考虑整个问题的另一种解决方法——我们可以将Time对象转换为整数，并利用计算机知道如何做整数运算的事实。

下面是一个将Time对象转换为整数的函数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

而下面是一个将整数转换回Time对象的函数（记着`divmod`函数将第一个参数除以第二个参数，并以元组的形式返回商和余数）：

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

你可能需要思考一下，并运行一些测试，来说服自己这些函数是正确的。一种测试它们的方法是对很多`x`值检查`time_to_int(int_to_time(x)) == x`。这是一致性检验的一个例子。

一旦确认它们是正确的，就可以使用它们重写`add_time`：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

这个版本比最初版本短得多，并且也很容易检验。作为练习，使用`time_to_int`和`int_to_time`重写`increment`函数。

从某个角度看，在六十进制和十进制之间来回转换比只处理时间更难。进制转换更加抽象；我们对时间值的直觉更好。

但如果我们将时间看作六十进制数，并做好了编写转换函数（`time_to_int`和`int_to_time`）的先期投入，就能得到一个更短，更可读，也更可靠的函数。

它也让我们今后更容易添加功能。例如，假设将两个Time对象相减来获得它们之间的时间间隔。简单的做法是使用借位实现减法。而使用转换函数则更简单，且更容易正确。

讽刺的是，有时候把一个问题弄得更难（或者更通用）反而会让它更简单（因为会有更少的特殊情况以及更少的出错机会）。

16.5　调试

一个Time对象当`minute`和`second`的值在0到60之间（包含0但不包含60）以及`hour`是正值时，是合法的。`hour`和`minute`应当是整数值，但我们也许需要允许`second`拥有小数值。

这些需求称为不变式，因为它们应当总是为真。换句话说，如果它们不为真，则一定有什么地方出错了。

编写代码来检查不变式可以帮你探测错误并找寻它们的根源。例

如，你可以写一个像valid_time这样的函数，接收Time对象，并在它违反了一个不变式时，返回False：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

接着在每个函数的开头，可以检查参数，确保它们是有效的：

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

或者可以使用一个assert语句。它会检查一个给定的不变式，并当检查失败时抛出异常：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assert语句很有用，因为它们区分了处理普通条件的代码和检查

错误的代码。

16.6 术语表

原型和补丁（prototype and patch）：一种开发计划模式，先编写程序的粗略原型，并测试，在找到错误时更正。

有规划开发（planned development）：一种开发计划模式，先对问题有了高阶的深入理解，并且比增量开发或者原型开发有更多的规划。

纯函数（pure function）：不修改任何形参对象的函数。大部分纯函数都有返回值。

修改器（modifier）：修改一个或多个形参对象的函数。大部分修改器都不返回值，也就是返回`None`。

函数式编程风格（functional programming style）：一种编程设计风格，其中大部分函数都是纯函数。

不变式（invariant）：在程序的执行过程中应当总是为真的条件。

`assert`语句（assert statement）：一种检查某个条件，如果检查失败则抛出异常的语句。

16.7 练习

本章中的代码示例可以从http://thinkpython2.com/code/Time1.py下载，这些练习的解答可以从http://thinkpython2.com/code/Time1_soln.py下

载。

练习**16-1**

编写一个函数`mul_time` 接收一个Time对象以及一个整数，返回一个新的Time对象，包含原始时间和整数的乘积。

然后使用`mul_time` 来编写一个函数，接收一个Time对象表示一场赛车的结束时间，以及一个表示距离的数字，并返回一个Time对象表达平均节奏（每英里花费的时间）。

练习**16-2**

`datetime` 模块提供了`time` 对象，和本章中的Time对象类似，但它们提供了更丰富的方法和操作符。在http://docs.python.org/3/library/datetime.html阅读相关文档。

1．使用`datetime` 模块来编写一个程序获取当前日期并打印出今天是周几。

2．编写一个程序接收生日作为输入，并打印出用户的年龄，以及到他们下一次生日还需要的天数、小时数、分钟数和秒数。

3．对于生于不同天的两个人，总有一天，一个人的年龄是另一个人的两倍。我们称这是他们的“双倍日”。编写一个程序接收两个生日，并计算出它们的“双倍日”。

4．再增加一点挑战，编写一个更通用的版本，计算一个人比另一个人大*n* 倍的日子。

解答：http://thinkpython2.com/code/double.py。

第17章　类和方法

虽然我们已经使用了Python的一些面向对象特性，但前两章的程序还算不上真正的面向对象，因为它们没有体现用户自定义类型之间的关联，以及操作它们的函数。下一步是将那些函数转换成方法，让这种关联更加明显。

本章的代码示例可以从http://thinkpython2.com/code/Time2.py下载，而本章练习的解答参见http://thinkpython2.com/code/Point2_soln.py。

17.1　面向对象特性

Python是一门面向对象编程语言，它提供了一些支持面向对象编程的语言特性，这些特性有如下明确的特征。

- 程序包括类定义和方法定义。
- 大部分计算都通过对象的操作来表达。
- 每个对象定义对应真实世界的某些对象或概念，而方法则对应真实世界中对象之间交互的方式。

例如，第16章中定义的`Time`类对应于人们记录一天中的时间的方式，而其中我们定义的函数对应于人们平时处理时间所做的事情。类似地，`Point`和`Rectangle`类对应于数学中点和矩形的概念。

目前为止，我们还没有利用上Python所提供的面向对象编程特性。

严格地说，这些特性并不是必需的；它们中大部分都是我们已经做过的事情的另一种选择方案。但在很多情况下，这种方案更简洁，更能准确地表达程序的结构。

例如，在`Time1.py` 程序中，类定义和接着的函数定义并没有明显的关联。稍加观察，很明显每个函数都至少接收一个Time对象作为参数。

这种现象就是方法 的由来。一个方法即是和某个特定类相关联的函数。我们已经见过字符串、列表、字典和元组的方法。本章中，我们会为用户定义类型定义方法。

方法和函数在语义上是一样的，但在语法上有两个区别。

- 方法定义写在类定义之中，更明确的表示类和方法的关联。
- 调用方法和调用函数的语法形式不同。

在接下来几节中，我们会将前两章中定义的函数转换为方法。这种转换是纯机械式的；你可以依照一系列步骤完成它。如果你能够轻松地在方法和函数之间转换，也就能够在任何情况下选择最适合的形式了。

17.2　打印对象

在第16章中，我们在练习16-1中定义了一个名为`Time` 的类，你写过一个名为`print_time` 的函数：

```
class Time:
    """Represents the time of day."""
```

```
def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

要调用这个函数，需要传入一个Time对象作为实参：

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

要把`print_time`转换为方法，我们只需要将函数定义移动到类定义中即可。注意缩进的改变。

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

现在有两种方式可以调用`print_time`。第一种（更少见的）方式是使用函数调用语法：

```
>>> Time.print_time(start)
09:45:00
```

在这里的点表示法中，Time 是类的名称，而print_time 是方法的名称。start 是作为参数传入的。

另一种（更简洁的）方式是使用方法调用语法：

```
>>> start.print_time()
09:45:00
```

在这里的点表示法中，print_time （又一次）是方法的名称，而start 是调用这个方法的对象，也称为主体 （subject）。和一句话中主语用来表示这句话是关于什么东西的一样，方法调用的主体表示这个方法是关于哪个对象的。

在方法中，主体会被赋值给第一个形参，所以本例中start 被赋值给time 。

依惯例来，方法的第一个形参通常叫作self ，所以print_time 通常写成这样的形式：

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

这种惯例的原因是一个隐喻。

- 函数调用的语法print_time(start) 暗示函数是活动主体。它仿佛在说：“喂，print_time！这里是一个让你打印的对象。”
- 在面向对象编程中，对象是活动主体。类似start.print_time() 的方法调用相当于说：“喂，start！请打印你自己。”

这种视角的改变可能变得更礼貌，但是否也更有用这一点却不那么明显。在我们已经见过的例子中，它也许并没有更有用。但有时候将函数的责任转到对象上，使我们能够编写功能更丰富的函数（或方法），也使代码的维护和复用更容易。

作为练习，将16.4节中的函数time_to_int 重写为方法。你大概也会想将int_to_time 重写为方法，但这么做实际上没有什么意义，因为你找不到可以调用它的对象。

17.3　另一个示例

下面是函数increment （参见16.3节）的另一个重写成了方法的版本：

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

这个版本假设time_to_int 已经写成了方法。另外，注意它是一

个纯函数，而不是一个修改器。

下面是调用increment的方式：

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

主体start 赋值给第一个形参self，实参1337，赋值给第二个形参seconds 。

这种机制有时也会带来困惑，尤其在当程序出错的时候。例如，如果使用两个实参调用increment，则会得到：

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

错误信息初看起来似乎很令人困惑，因为括号里只有两个实参。但调用的主体也被看作一个实参，所以其实总共有3个。

另外，按位实参（positional argument）指的是没有指定名称的实参，也就是说，它不是一个关键词实参。在下面这个函数调用中，parrot 和cage 是按位实参，而dead 是一个关键词实参：

```
sketch(parrot, cage, dead=True)
```

17.4　一个更复杂的示例

重写函数is_after （见16.1节）稍微更复杂一些，因为它接收两个Time对象作为形参。这种情形下，依惯例，第一个形参命名为self，而第二个形参命名为other：

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

要使用这个方法，需要在一个对象上调用它，并传入另一个对象作为实参：

```
>>> end.is_after(start)
True
```

这种语法的一个好处是，阅读起来几乎和英语一样：“end is after start?”。

17.5　init 方法

init 方法（即“initialization”的简写，意思是初始化）是一个特殊方法，当对象初始化时会被调用。它的全名是__init__ （两个下划线，接着是init ，再接着两个下划线）。Time 类的init 方法可能如下所示：

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

__init__ 的形参和类的属性名称常常是相同的。语句

```
self.hour = hour
```

将形参hour 的值存储为self 的一个属性。

形参是可选的，所以当你不使用任何实参调用Time 时，会得到默认值：

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

如果提供1个实参，它会覆盖hour：

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

如果提供2个实参，它会覆盖hour 和minute：

```
>>> time = Time (9, 45)
>>> time.print_time()
09:45:00
```

如果提供3个实参，它们会覆盖全部3个默认值。

作为练习，为Point 类编写一个init 方法，接收x 和y 作为可选形参，并将它们的值赋到对应的属性上。

17.6 __str__ 方法

__str__ 和__init__ 类似，是一个特殊方法，它用来返回对象的字符串表达形式。

例如，下面是一个Time对象的str 方法：

```
# inside class Time:

    def __str__(self):
```

```
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

当你打印对象时，Python会调用str 方法。

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

当我编写一个新类时，我总是开始先写`__init__`，以便初始化对象，然后会写`__str__`，以便调试。

作为练习，为Point 类编写一个str 方法。创建一个Point对象并打印它。

17.7　操作符重载

通过定义其他的特殊方法，你可以为用户定义类型的各种操作符指定行为。例如，如果你为Time 类定义一个`__add__` 方法，则可以在Time对象上使用+ 操作符。

下面是这个方法的定义：

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

```

```

而下面是如何使用它：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

当你对Time对象应用+操作符时，Python会调用`__add__`。当你打印结果时，Python会调用`__str__`。幕后其实发生了很多事情！

修改操作符的行为以便它能够作用于用户定义类型，这个过程称为操作符重载。对每一个操作符，Python都提供了一个对应的特殊方法，如`__add__`。更多细节，可以参见 http://docs.python.org/3/reference/datamodel.html#specialnames。

作为练习，为Point类编写一个`add`方法。

17.8　基于类型的分发

在前面一节中我们将两个Time对象相加，但你也可能会想要将一个Time对象加上一个整数。接下来是`__add__`的一个版本，检查`other`的类型，并调用`add_time`或`increment`：

```
# inside class Time:
```

```
    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

内置函数isinstance 接收一个值与一个类对象，并当此值是此类的一个实例时返回True 。

如果other 是一个Time对象，__add__ 会调用add_time 。否则它认为实参是整数，并调用increment 。这个操作称为基于类型的分发（type-based dispatch），因为它根据形参的类型，将计算分发到不同的方法上。

下面是使用不同类型的实参调用+ 操作符的示例：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

遗憾的是，这个加法的实现并不满足交换律。如果整数是第一个操作数，则会得到：

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

问题在于，这里和之前询问一个Time对象加上一个整数不同，Python在询问一个整数去加上一个Time对象，而它并不知道如何去做到。但这个问题也有一个聪明的解决方案：特别方法`__radd__`，意即“右加法”（right-side add）。当Time对象出现在+ 号的右侧时，会调用这个方法。下面是它的定义：

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

而下面是如何使用：

```
>>> print(1337 + start)
10:07:17
```

作为练习，为Point类编写一个`add` 方法，可以接收一个Point对象或者一个元组。

- 如果第二个操作对象是一个Point对象，则方法应该返回一个新的Point对象，其x 坐标是两个操作对象的x 坐标的和，y 坐标也是类似。
- 如果第二个操作对象是一个元组，方法则将第一个元素和x 坐标相加，将第二个元素和y 坐标相加，并返回一个包含相加结果的新Point对象。

17.9　多态

当需要时，基于类型的分发很有用，但（幸运的是）我们并不总是需要它。通常可以编写函数处理不同类型的参数来避免它。

我们编写的很多处理字符串的函数，实际上对其他序列类型也可以用。例如，在11.1节中，我们使用`histogram`来记录单词中每个字母出现的次数：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

这个函数对列表、元组甚至是字典都可用，只要`s`的元素是可散列的，因而可以用作`d`的键即可：

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

处理多个类型的函数称为多态（polymorphic）。多态可以促进代码复用。例如，用来计算一个序列所有元素的和的内置函数sum，对所有其元素支持加法的序列都可用。

由于Time对象提供了add 方法，它们也可以使用sum：

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print (total)
23:01:00
```

总的来说，如果函数内部所有的操作都支持某种类型，那么这个函数就可以用于那种类型。

当你发现一个写好的函数，竟然有出人意料的效果，可以用于没有计划过的类型时，这才是最好的多态。

17.10　接口和实现

面向对象设计的目标之一是提高软件的可维护性，也就是说，当系

统的其他部分改变时，程序还能够保持正确运行，并且能够修改程序来适应新的需求。

将接口和实现分离的设计理念，可以帮我们更容易达到这个目标。对于对象来说，那意味着类所提供的方法应该不依赖于其属性的表达方式。

例如，在本章中我们开发了一个类来表示一天中的时间。这个类提供的方法包括`time_to_int`、`is_after`和`add_time`。

我们可以使用几种不同的方式来实现这些方法。实现的细节依赖于我们表达时间概念的方式。在本章中，Time对象的属性是`hour`、`minute`和`second`。

用另一种方案，我们可以将这些属性替换成一个整数，表示从凌晨开始到现在的秒数。这种实现可能会让一些方法，如`is_after`，更容易实现，但也会让另一些方法更难实现。

在部署一个新类时，你可能会发现更好的实现。如果程序中其他部分用到你的类，则修改接口会非常消耗时间，并且容易产生错误。

但是，如果很谨慎小心地设计接口，则可以在不修改接口的情况下修改实现，这样程序的其他部分就不需要跟着修改。

17.11　调试

在程序运行的任何时刻，往对象上添加属性都是合法的，但如果遵

守更严格的类型理论，让对象拥有相同的类型却有不同的属性组，会很容易导致错误。通常来说，在init 方法中初始化对象的全部属性是个好习惯。

如果并不清楚一个对象是否拥有某个属性，可以使用内置函数hasattr （参见15.7节）。

另一种访问一个对象的属性的方法是使用内置函数vars ，它接收一个对象，并返回一个将属性名称（字符串形式）映射到属性值的字典对象：

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

为了调试，你可能会发现将这个函数放在手边是很有用的：

```
def print_attributes(obj):
    for attr in vars(obj):
        print (attr, getattr(obj, attr))
```

print_attributes 遍历对象的属性字典，并打印出每个属性的名称和相应的值。

内置函数getattr 接收一个对象以及一个属性名称（字符串形式）并返回属性的值。

17.12　术语表

面向对象语言（object-oriented language）：一种提供诸如用户定义类型和方法之类的语言特性，以方便面向对象编程的语言。

面向对象编程（object-oriented programming）：一种编程风格，数据和修改数据的操作组织成类和方法的形式。

方法（method）：在类定义之内定义的函数，在类的实例上调用。

主体（subject）：调用方法所在的对象。

按位实参（positional argument）：一个不包含参数名字的实参，所以它不是一个关键词实参。

操作符重载（operator overloading）：修改一个类似+号这样的操作符的行为，使之可以用于用户定义类型。

基于类型的分发（type-based dispatch）：一种编程模式，检查操作对象的类型，并对不同类型调用不同的函数。

多态（polymorphic）：函数的一种属性，可以处理多种类型的参数。

信息隐藏（information hiding）：对象提供的接口不应当依赖于其实现，特别是其属性的表达形式的原则。

17.13 练习

练习**17-1**

从http://thinkpython2.com/code/Time2.py 下载本章的代码。将`Time`的属性改为从凌晨开始到现在的秒数。接着修改方法（以及函数`int_to_time`），以适应新的属性实现。你应该不需要修改`main`里面的测试代码。当你做完之后，输出应该和以前一样。

解答：http://thinkpython2.com/code/Time2_soln.py

练习**17-2**

这个练习提醒你关于Python的一种最常见且最难查找的错误的故事。

编写一个叫作`Kangaroo`（袋鼠）的类，有如下方法。

1．一个`__init__`方法，将属性`pouch_contents`（口袋中的东西）初始化为一个空列表。

2．一个`put_in_pouch`方法，接收任何类型的对象，并将它添加到`pouch_contents`中。

3．一个`__str__`方法，返回Kangaroo对象以及口袋中的内容的字符串表达形式。

创建两个`Kangaroo`对象，将它们赋值到变量`kanga`和`roo`，并

将roo 添加到kanga 的口袋中。

下载http://thinkpython.com/code/BadKangaroo.py，它包含了前面问题的解答，但里面有一个很大很丑陋的bug。找出并修复这个bug。

如果你遇到阻碍，可以下载http://thinkpython.com/code/GoodKangaroo.py，它解释了问题的原因，并提供了一个解决方案。

第18章　继承

和面向对象编程最常相关的语言特性就是继承（**inheritance** ）。继承指的是根据一个现有的类型，定义一个修改版本的新类的能力。本章中我会使用几个类来表达扑克牌、牌组以及扑克牌型，用于展示继承特性。

如果你不玩扑克，可以在http://en.wikipedia.org/wiki/Poker里阅读相关介绍，但其实并不必要；我会在书中介绍练习中所需知道的东西。

本章的代码示例可以从http://thinkpython2.com/code/Card.py下载。

18.1　卡片对象

一副牌里有52张牌，共有4个花色，每种花色13张，大小各不相同。花色有黑桃（Spade）、红桃（Heart）、方片（Diamond）和草花（Club）（在桥牌中，这几个花色是降序排列的）。每种花色的13张牌分别为：Ace、2、3、4、5、6、7、8、9、10、Jack、Queen和King。根据你玩的不同游戏，Ace可能比King大，也可能比2小。

如果我们定义一个新对象来表示卡牌，则其属性显然应该是`rank`（大小）和`suit` （花色）。但属性的值就不那么直观了。一种可能是使用字符串，例如，用`'Spade'` 表示花色，用`'Queen'` 表示大小。这种实现的问题之一是比较大小和花色的高低时会比较困难。

另一种方案是使用整数来给大小和花色编码 。在这个语境中，“编码”意味着我们要定义一个数字到花色，或者数字到大小的映射。这种编码并不意味着它是秘密（那样就应该称为“加密”了）。

例如，下表展示了花色和对应的整数编码：

```
Spades    →  3
Hearts    →  2
Diamonds  →  1
Clubs     →  0
```

这个编码令我们可以很容易地比较卡牌；因为更大的花色映射到更大的数字上，我们可以直接使用编码来比较花色。

卡牌大小的映射相当明显；每个数字形式的大小映射到相应的整数上，而对于花牌：

```
Jack   →   11
Queen  →   12
King   →   13
```

我使用“→”符号，是为了说明这些映射并不是Python程序的一部分。它们是程序设计的一部分，但并不在代码中直接表现。

Card 类的定义如下：

```
class Card:
```

```
    """Represents a standard playing card."""

    def _ _init_ _(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

和前面一样，init 方法对每个属性定义一个可选形参。默认的卡牌是草花2。

要创建一个Card对象，使用你想要的花色和大小调用Card：

```
queen_of_diamonds = Card(1, 12)
```

18.2 类属性

为了能将Card对象打印成人们容易阅读的格式，我们需要将整数编码映射成对应的大小和花色。自然的做法是使用字符串列表。我们将这些列表赋到类属性 上：

```
# 在Card类里:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']

    def _ _str_ _(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

suit_names 和rank_names 这样的变量，定义在类之中，但在任何方法之外，我们称为类属性。因为它们是和类对象Card 相关联的。

这个术语和suit 与rank 之类的变量相区别。那些称为实例属性，因为它们是和一个特定的实例相关联的。

两种属性都使用句点表示法访问。例如，在__str __中，self 是一个Card对象，而self.rank 是它的大小。相似地，Card 是一个类对象，而Card.rank_names 是关联到这个类的一个字符串列表。

每个卡片都有它自己的suit 和rank ，但总共只有一个suit_names 和rank_names 。

综合起来，表达式Card.rank_names[self.rank] 意思是“使用对象self 的属性rank 作为索引，从类Card 的列表rank_names 中选择对应的字符串”。

rank_names 的第一个元素是None ，因为没有大小为0的卡牌。因为使用None 占据了一个位置，我们就可以得到从下标2到字符串'2' 这样整齐的映射。如果要避免这种操作，可以使用字典而不是列表。

利用现有的方法，可以创建并打印卡牌：

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

图18-1展示了Card 类对象和一个Card实例。Card 是一个类对象，所以它的类型是type 。card1 的类型是Card 。为了节省空间，我没有画出suit_names 和rank_names 的内容。

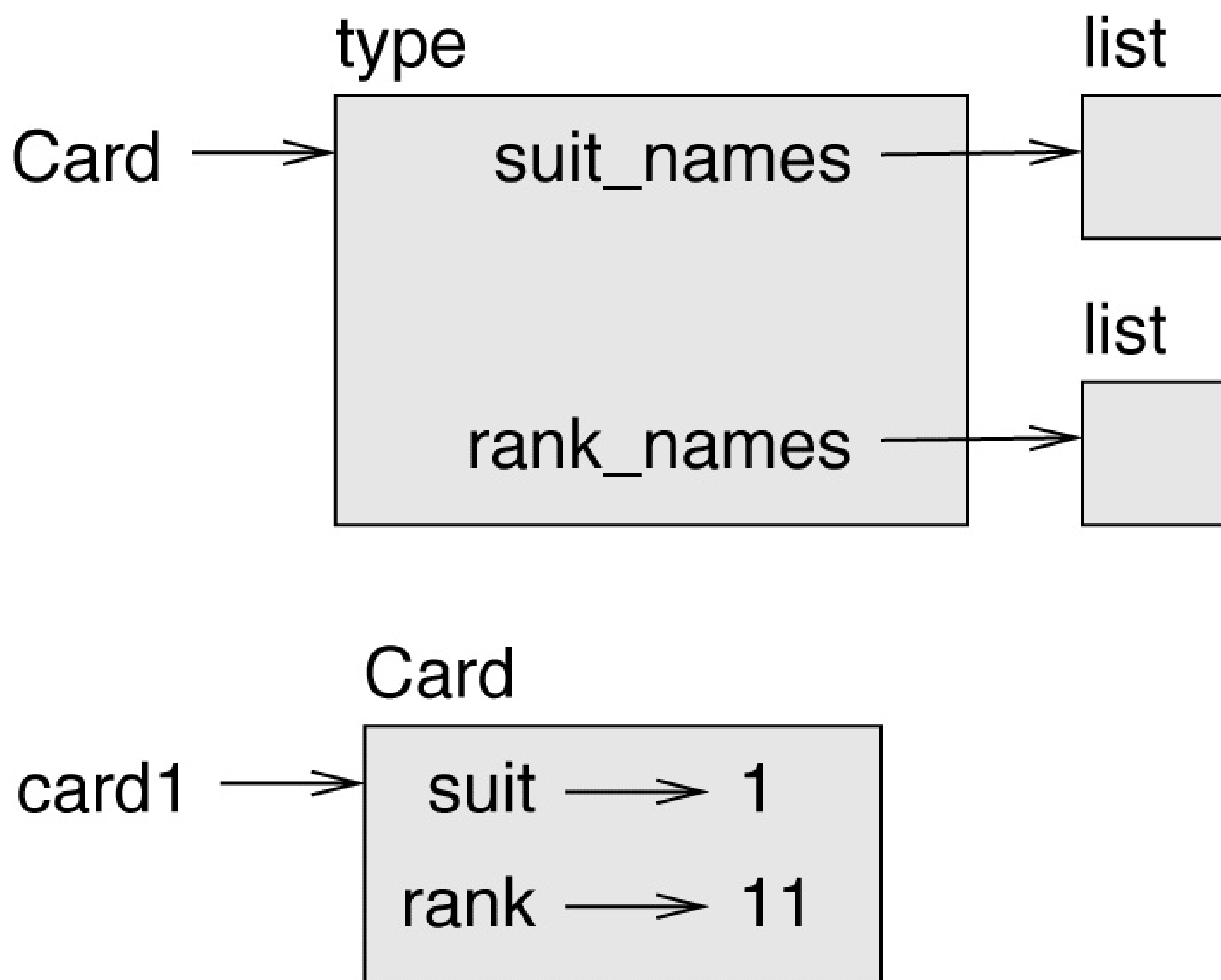


图18-1　对象图

18.3　对比卡牌

对于内置类型，我们用比较操作符（< 、> 、== 等）来比较对象并

决定哪一个更大、更小或者相等。对于用户定义类型，我们可以通过提供一个方法__lt __，代表“less than”,来重载内置操作符的行为。

__lt __接收两个形参，self 和other ，当第一个对象严格小于第二个对象时返回True 。

卡牌的正确顺序并不显而易见。例如，草花3和方片2哪个更大？一个牌面数大，另一个花色大。为了比较卡牌，需要决定大小和花色哪个更重要。

这个问题的答案取决于你在玩哪种牌类游戏，但为了简单起见，我们随意做一个决定，认为花色更重要，于是，所有的黑桃比所有的方片都大，依此类推。

这一点决定后，我们就可以编写__lt __函数：

```
# 在 Card类里:

    def _ _lt_ _(self, other):
        # 检查花色
         if self.suit < other.suit: return True
         if self.suit > other.suit: return False

         # 花色相同，检查大小
         return self.rank < other.rank
```

使用元组比较，可以写得更紧凑：

```
# 在Card类里:

    def _ _lt_ _(self, other):
```

```
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

作为练习，为时间对象编写一个__lt __方法。你可以使用元组比较，也可以考虑使用整数比较。

18.4 牌组

现在我们已经有了卡牌（card），下一步就是定义牌组（deck）。由于牌组是由卡牌组成的，很自然地，每个Deck对象应该有一个属性包含卡牌的列表。

下面是Deck 的类定义。init 方法创建cards 属性，并生成52张牌的标准牌组：

```
class Deck:

    def _ _init_ _(self):
        self.cards = []
        for suit in range(4):
             for rank in range(1, 14):
                  card = Card(suit, rank)
                  self.cards.append(card)
```

填充牌组最简单的办法是使用嵌套循环。外层循环从0到3遍历各个花色。内层循环从1到13遍历卡牌大小。每次迭代使用当前的花色和大

小创建一个新的Card对象，并将它添加到self.cards 中。

18.5 打印牌组

下面是Deck 的一个__str__ 方法：

```
# 在 Deck类里:

    def __str__(self):
        res=[]
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

这个方法展示了一种累积构建大字符串的方法：先构建一个字符串的列表，再使用字符串方法join 。内置函数str 会对每个卡牌对象调用__str__ 方法并返回字符串表达形式。

由于我们对一个换行符调用join 函数，卡片之间用换行分隔。下面是打印的结果：

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

虽然结果显示了52行，它仍然是一个包含换行符的字符串。

18.6　添加、删除、洗牌和排序

为了能够发牌，我们需要一个方法从牌组中抽取一张牌并返回。列表方法pop 为此提供了一个方便的功能：

```
# 在 Deck类里:

    def pop_card(self):
        return self.cards.pop()
```

由于pop 从列表中抽出最后 一张牌，我们其实是从牌组的底端发牌的。

要添加一个卡牌，我们可以使用列表方法append：

```
# 在 Deck类里:

    def add_card(self, card):
        self.cards.append(card)
```

像这样调用另一个方法，却不做其他更多工作的方法，有时候称为一个饰面 （veneer）。这个比喻来自于木工行业，在木工行业中饰面是

为了改善外观而粘贴到便宜的木料表面的薄薄的一层优质木料。

在这个例子里，add_card 是一个“薄薄”的方法，用更适合牌组的术语来表达一个列表操作。它改善了实现的外观（或接口）。

作为另一个示例，我们可以使用random 模块的函数shuffle 来编写一个Deck方法shuffle （洗牌）：

```
# 在 Deck类里:

    def shuffle(self):
        random.shuffle(self.cards)
```

不要忘记导入random 模块。

作为练习，编写一个Deck方法sort ，使用列表方法sort 来对一个Deck 中的卡牌进行排序。sort 使用我们定义的__lt __方法来决定顺序。

18.7 继承

继承是一种能够定义一个新类对现有的某个类稍作修改的语言特性。作为示例，假设我们想要一个类来表达一副“手牌”，即玩家手握的一副牌。一副手牌和一套牌组相似：都是由卡牌的集合组成，并且都需要诸如增加和移除卡牌的操作。

一副手牌和一套牌组也有区别；我们期望手牌拥有的一些操作，对

牌组来说并无意义。例如，在扑克牌中，我们可能想要比较两副手牌来判断谁获胜了。在桥牌中，我们可能需要为一副手牌计算分数以叫牌。

这种类之间的关系——相似，但不相同——让它成为继承。要定义一个继承现有类的新类，可以把现有类的名称放在括号之中：

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

这个定义说明Hand 从Deck 继承而来；这意味着我们可以像Deck对象那样在Hand对象上使用pop_card 和add_card 方法。

当新类继承现有类时，现有的类被称为父类 （parent），而新类则称为子类 （child）。

在本例中，Hand 也会继承Deck 的__init __方法，但它和我们想要的并不一样：我们不需要填充52张卡牌，Hand的init 方法应当初始化cards 为一个空列表。

如果我们为Hand 类提供一个init 方法，它会覆盖Deck 类的方法：

```
# 在Hand类里:

    def _ _init_ _(self, lable=''):
        self.cards = []
        self.label = label
```

在创建Hand对象时，Python会调用这个init方法而不是Deck中的那个：

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

其他的方法是从Deck中继承而来的，所以我们可以使用pop_card和add_card来出牌：

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

下一步很自然地就是将这段代码封装起来成为一个方法move_cards：

```
# 在 Deck类里:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` 接收两个参数，一个Hand对象以及需要出牌的牌数。它会修改`self` 和`hand` ，返回`None` 。

有的情况下，卡牌会从一副手牌中移除转入到另一幅手牌中，或者从手牌中回到牌组。你可以使用`move_cards` 来处理全部这些操作：`self` 既可以是一个Deck对象，也可以是一个Hand对象。而`hand` 参数，虽然名字是`hand` ，却也可以是一个Deck对象。

继承是很有用的语言特性。有些程序不用继承写，会有很多重复代码，使用继承后就会更加优雅。继承也能促进代码复用，因为你可以在不修改父类的前提下对它的行为进行定制化。有的情况下，继承结构反映了问题的自然结构，所以也让设计更容易理解。

但另一方面，继承也可能会让代码更难读。有时候当一个方法被调用时，并不清楚到哪里能找到它的定义。相关的代码可能散布在几个不同的模块中。并且，很多可以用继承实现的功能，也能不用它实现，甚至可以实现得更好。

18.8　类图

至此我们已见过用于显示程序状态的栈图，以及用于显示对象的属性和属性值的对象图。这些图表展示了程序运行中的一个快照，所以当程序继续运行时它们会跟着改变。

它们也极其详细；在某些情况下，是过于详细了。而类图对程序结

构的展示相对来说更加抽象。它不会具体显示每个对象，而是显示各个类以及它们之间的关联。

类之间有下面几种关联。

- 一个类的对象可能包含其他类的对象的引用。例如，每个Rectangle对象都包含一个到Point对象的引用，而每一个Deck对象包含到很多Card对象的引用。这种关联称为**HAS-A** （有一个），也就是说，“矩形（Rectangle）中有一个点（Point）”。
- 一个类可能继承自另一个类。这种关系称为**IS-A** （是一个），也就是说，“一副手牌（Hand）是一个牌组（Deck）”。
- 一个类可能依赖于另一个类，也就是说，一个类的对象接收另一个类的对象作为参数，或者使用另一个类的对象来进行某种计算。这种关系称为依赖 （dependency）。

类图 用图形展示了这些关系。例如，图18-2展示了`Card` 、`Deck` 和`Hand` 之间的关系。

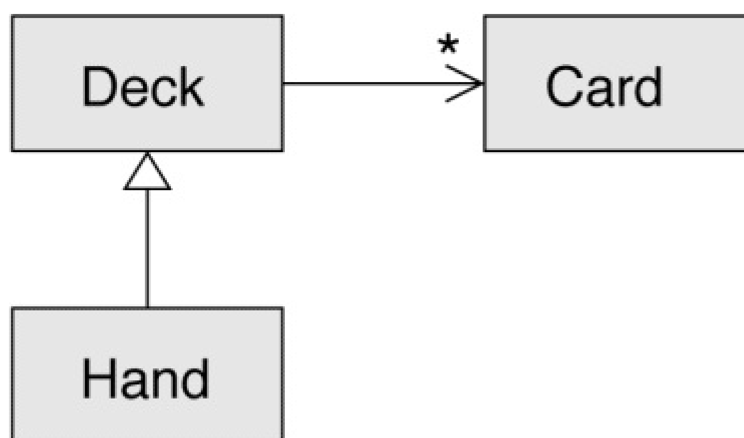


图18-2　类图

空心三角形箭头的线代表着一个IS-A关系；这里表示Hand是继承自Deck的。

标准的箭头表示HAS-A关系；这里表示Deck对象中有到Card对象的引用。

箭头附近的星号（*）表示是关联重数标记；它表示Deck中有多少Cards。这个数可以是一个简单的数字，如52，或者一个范围，如5..7，或者一个星号，表示Deck可以有任意数量的Card引用。

图18-2中没有任何依赖关系。依赖关通常使用虚线箭头表示。或者，如果有太多的依赖，有时候会忽略它们。

更详细的图可能会显示出Deck对象实际上包含了一个Card的列表。但在类图中，像列表、字典这样的内置类型通常是不显示的。

18.9　数据封装

前几章展示了一个我们可以称为“面向对象设计”的开发计划。我们发现需要的对象，如Point、Rectangle和Time，并定义类来表达它们。每个类都是一个对象到现实世界（或者最少是数学世界）中的某种实体的明显对应。

但有时候你到底需要哪些对象、它们如何交互，并不那么显而易见。这时候你需要另一种开发计划。和之前我们通过封装和泛化来发现函数接口的方式相同，我们可以通过数据封装 来发现类的接口。

13.8节提供了一个很好的示例。如果从http://thinkpython2.com/code/markov.py下载我的代码，你会发现它使用了两个全局变量（suffix_map 和prefix ）并且在多个函数中进行读写。

```
suffix_map = {}
prefix = ()
```

因为这些变量是全局的，我们每次只能运行一个分析。如果我们读入两个文本，它们的前缀和后缀就会添加到相同的数据结构中（最后可以用来产生一些有趣的文本）。

若要多次运行分析，并保证它们之间的独立，我们可以将每次分析的状态信息封装成一个对象。下面是它的样子：

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

接下来，我们将那些函数转换为方法。例如，下面是process_word：

```
def process_word(self, word, order=2):
     if len(self.prefix) < order:
          self.prefix += (word,)
```

```
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # 如果这个前缀不存在，创建一项
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

像这样转换程序——修改设计但不修改其行为——是重构（参见4.7节）的另一个示例。

这个例子给出了一个设计对象和方法的开发计划。

1．从编写函数、（如果需要的话）读写全局变量开始。

2．一旦你的程序能够正确运行，查看全局变量与使用它们的函数的关联。

3．将相关的变量封装成为对象的属性。

4．将相关的函数转换为这个新类的方法。

作为练习，从http://thinkpython2.com/code/markov.py下载我的Markov代码，并按照上面描述的步骤将全局变量封装为一个叫作`Markov` 的新类的属性。

解答：http://thinkpython2.com/code/Markov.py（注意M是大写的）。

18.10　调试

继承会给调试带来新的挑战，因为当你调用对象的方法时，可能无法知道调用的到底是哪个方法。

假设你在编写一个操作Hand对象的函数。你可能希望能够处理所有类型的Hand，如PokerHands、BridgeHands等。如果你调用一个方法，如`shuffle`，可能调用的是`Deck`中定义的方法，但如果任何子类重载了这个方法，则你调用的会是那个重载的版本。

一旦你无法确认程序的运行流程，最简单的解决办法是在相关的方法开头添加一个打印语句。如果`Deck.shuffle`打印一句`Running Deck.shuffle`这样的信息，则当程序运行时会跟踪运行的流程。

或者，你也可以使用下面这个函数。它接收一个对象和一个方法名（字符串形式），并返回提供这个方法的定义的类：

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty. _ _dict_ _:
            return ty
```

下面是使用的示例：

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

所以这个Hand对象的`shuffle`方法是在`Deck`类中定义的那个。

`find_defining_class`使用`mro`方法来获得用于搜索调用方法的类对象（类型）列表。“MRO”意思是“method resolution order”（方法查找顺序），是Python解析方法名称的时候搜索的类的顺序。

一个设计建议：每次重载一个方法时，新方法的接口应当和旧方法的一致。它应当接收相同的参数，返回相同的类型，并服从同样的前置条件与后置条件。如果遵循这个规则，你会发现任何为如Deck这样的父类设计的函数，都可以使用Hand或PokerHand这样的子类的实例。

如果你破坏这个也称为“Liskov替代原则”的规则，你的代码可能会像一堆（不好意思）纸牌屋一样崩塌。

18.11　术语表

编码 （encode）：使用一个集合的值来表示另一个集合的值，需要在它们之间建立映射。

类属性 （class attribute）：关联到类对象上的属性。类属性定义在类定义之中，但在所有方法定义之外。

实例属性 （instance attribute）：和类的实例关联的属性。

饰面 （veneer）：一个方法或函数，它调用另一个函数，却不做其他计算，只是为了提供不同的接口。

继承（inheritance）：可以定义一个新类，它是一个现有的类的修改版本。

父类（parent class）：被子类所继承的类。

子类（child class）：通过继承一个现有的类来创建的新类，也叫作“subclass”。

IS-A关联（IS-A relationship）：子类与父类之间的关联。

HAS-A关联（HAS-A relationship）：两个类之间的一种关联：一个类包含另一个类的对象的引用。

依赖（dependency）：两个类之间的一种关联。一个类的实例使用另一个类的实例，但不把它们作为属性存储起来。

类图（class diagram）：用来展示程序中的类以及它们之间的关联的图。

重数（multiplicity）：类图中的一种标记方法，对于HAS-A关联，用来表示一个类中有多少对另一个类的对象的引用。

数据封装（data encapsulation）：一个程序开发计划。先使用全局变量来进行原型设计，然后将全局变量转换为实例属性做出最终版本。

18.12　练习

练习**18-1**

针对下面的程序，画一张UML类图，展示这些类以及它们之间的关联：

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong

class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
         self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

练习18-2

编写一个名为`deal_hands`的Deck方法，接收两个形参：手牌的数量以及每副手牌的牌数。它会根据形参创建新的Hand对象，按照每副手牌的牌数出牌，并返回一个Hand对象列表。

练习18-3

下面列出的是扑克牌中可能的手牌，按照牌值大小的增序（也是可能性的降序）排列。

- 对子 （pair）：两张牌大小相同。
- 两对 （two pair）：两个对子。
- 三条 （three of a kind）：三张牌大小相同。
- 顺子 （straight）：五张大小相连的牌（Ace既可以是最大也可以是最小，所以Ace-2-3-4-5是顺子，10-Jack-Queen-King-Ace也是，但Queen-King-Ace-2-3不是）。
- 同花 （flush）：五张牌花色相同。
- 满堂红 （full house）：三张牌大小相同，另外两张牌大小相同。
- 四条 （four of a kind）：四张牌大小相同。
- 同花顺 （straight flush）：顺子（如上面的定义）里的五张牌都是花色相同的。

本练习的目标是预测这些手牌的出牌概率。

1. 从http://thinkpython2.com/code下载这些文件。

- `Card.py`：本章中介绍的`Card`、`Deck`和`Hand`类的完整代码。
- `PokerHand.py`：表达扑克手牌的一个类，实现并不完整，包含一些测试它的代码。

2. 如果你运行`PokerHand.py`，它会连出7组包含7张卡片的扑克手牌，并检查其中有没有顺子。在继续之前请仔细阅读代码。

3. 在`PokerHand.py`中添加方法`has_pair`、`has_twopair`等。它们根据手牌是否达到相对应的条件来返回True或False。你的代码应当对任意数量的手牌都适用（虽然最常见的手牌数是5或7）。

4. 编写一个函数`classify`（分类），它可以弄清楚一副手牌中出现的最大的组合，并设置`label` 属性。例如，一副7张牌的手牌可能包含一个顺子以及一个对子；它应当标记为“flush”（顺子）。

5. 当你确保分类方法可用时，下一步是预测各种手牌的概率。在`PokerHand.py` 中编写一个函数，对一副牌进行洗牌，将其分成不同手牌，对手牌进行分类，并记录每种分类出现的次数。

6. 打印一个表格，展示各种分类以及它们的概率。更多次地运行你的程序，直到输出收敛到一个合理程度的正确性为止。将你的结果和http://en.wikipedia.org/wiki/ Hand_rankings上的值进行对比。

解答：http://thinkpython2.com/code/PokerHandSoln.py。

第19章　Python拾珍

本书的一大目标一直是尽可能少地介绍Python语言。如果做某种事情有两种方法，我会选择一种，并避免提及另一种。或者有时候，我会把另一种方法作为练习进行介绍。

本章我会带领大家回顾那些遗漏的地方。Python提供了不少并不是完全必需的功能（不用它们也能写出好代码），但有时候，使用这些功能可以写出更简洁、更可读或者更高效的代码，甚至有时候三者兼得。

19.1　条件表达式

我们在5.4节中见过条件语句。条件语句通常用来从两个值中选择一个。例如：

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

这条语句检查x 是否为正数。如果为正数，则计算math.log；如果为负数，math.log 会抛出ValueError 异常。为了避免程序停止，我们直接生成一个“NaN”，一个特殊的浮点数，代表“不是数”（Not A Number）。

我们可以用条件表达式 来更简洁地写出这条语句：

```
y = math.log(x) if x > 0 else float('nan')
```

这条语句几乎可以用英语直接读出来："y gets log-x if x is greater than 0; otherwise it gets NaN"（Y 的值在x 大于0 时是math.log(x)，否则是NaN ）。

递归函数有时候可以用条件表达式重写。例如，下面是factorial的一个递归版本：

```
def factorial(n):
     if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

我们可以将其重写为：

```
def factorial(n):
     return 1 if n == 0 else n * factorial(n-1)
```

条件表达式的另一个用途是处理可选参数。例如，下面是GoodKangaroo的init方法（参见练习17-2）：

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    self.pouch_contents = contents
```

我们可以将其重写为：

```
def __init__(self, name, contents=None):
     self.name = name
     self.pouch_contents = [] if contents == None else contents
```

一般来说，如果条件语句的两个条件分支都只包含简单的返回或对同一变量进行赋值的表达式，那么这个语句可以转化为条件表达式。

19.2　列表理解

在10.7节中我们已经见过映射和过滤模式。例如，下面的函数接收一个字符串列表，将每个元素通过字符串方法capitalize 进行映射，并返回一个新的字符串列表。：

```
def capitalize_all(t):
     res = []
     for s in t:
         res.append(s.capitalize())
     return res
```

我们可以用列表理解（list comprehension）把这个函数写得更紧凑：

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

上面的方括号操作符说明我们要构建一个新列表。方括号之内的表达式指定了列表的元素，而for 子句则表示我们要遍历的序列。

列表理解的语法有一点粗糙的地方，因为里面的循环变量，即本例中的s ，在表达式中出现在定义之前。

列表理解也可以用于过滤操作。例如，下面的函数选择列表t 中的大写元素，并返回一个新列表：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper()
            res.append(s)
    return res
```

我们可以用列表理解将其重写为：

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

对于简单表达式来说，列表理解更紧凑、更易于阅读，并且它们通常都比实现相同功能的循环更快，有时候甚至快很多。因此，如果你因为我没有早些提到它而恼怒，我表示十分理解。

但是我得辩解一下，列表理解更难以调试，因为你没法在循环内添加打印语句。我建议你只在计算简单到一次就能弄对的时候才使用它。对于初学者来说，这意味着从来不用。

19.3　生成器表达式

生成器表达式（generator expression）和列表理解类似，但是它使用圆括号，而不是方括号：

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

结果是一个生成器对象，它知道该如何遍历值的序列。但它又和列表理解不同，它不会一次把结果都计算出来，而是等待请求。内置函数`next`会从生成器中获取下一个值：

```
>>> next(g)
0
>>> next(g)
1
```

当到达序列的结尾后，next 会抛出一个StopIteration 异常。可以使用for 循环来遍历所有值：

```
>>> for val in g:
...     print(val)
4
9
16
```

生成器对象会跟踪记录访问序列的位置，所以for 循环会从上一个next 所在的位置继续。一旦生成器遍历结束，再访问它就会抛出StopException：

```
>>> next(g)
StopIteration
```

生成器表达式经常和sum 、max 和min 之类的函数配合使用：

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any 和all

Python提供了一个内置函数any，它接收一个由布尔值组成的序列，并在其中任何值是True时返回True。它可以用于列表：

```
>>> any([False, False, True])
True
```

但它更常用于生成器表达式：

```
>>> any(letter == 't' for letter in 'monty')
True
```

上面这个例子用处不大，因为它做的事情和in表达式一样。但是我们可以用any来重写9.3节中的搜索函数。例如，我们可以将avoids函数重写为：

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

这个函数读起来几乎和英语一致："word avoids forbidden if there are not any forbidden letters in word"（我们说一个word避免被禁止，是指word中没有任何被禁的字母）。

Python还提供了另一个内置函数all，它在序列中所有元素都

是True 时返回True 。作为练习，请使用all 重写9.3节中的uses_all 函数。

19.5　集合

我曾在13.6节中使用字典来寻找在文档中出现但不属于一个单词列表的单词。我写的函数接收一个字典参数d1 ，其中包含文档中所有的单词作为键；以及另一个参数d2 ，包含单词列表。它返回一个字典，包含d1 中所有不在d2 之中的键：

```
def substract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

在这些字典中，值都是None ，因为我们从来不用它们。因此，我们实际上浪费了一些存储空间。

Python还提供了另一个内置类型，称为集合（set ），它表现得和没有值而只使用键集合的字典类似。向一个集合添加元素很快，检查集合成员也很快。集合还提供方法和操作符来进行常见的集合操作。

例如，集合减法可以使用方法difference 或者操作符‘-’ 来实现。因此我们可以将substract 函数重写为：

```
def substract(d1, d2):
```

```
    return set(d1) - set(d2)
```

结果是一个集合而不是字典，但是对于遍历之类的操作，表现是一样的。

本书中的一些练习可以用集合来更加简洁且高效地实现。例如，练习10-7中的`has_duplicates` 函数，下面是使用字典来实现的一个解答：

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

一个元素第一次出现的时候，把它加入到字典中。如果相同的元素再次出现时，函数就返回`True` 。

使用集合，我们可以这样写同一个函数：

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

一个元素在一个集合中只能出现一次，所以如果t 中间的某个元素出现超过一次，那么变成集合后其长度会比t 小。如果没有任何重复元素，那么集合的长度应当和t 相同。

我们也可以使用集合来解决第9章中的一些练习。例如，下面是uses_only 函数使用循环来实现的版本：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

uses_only 检查word 中所有的字符是不是在available 中出现。我们可以这样重写：

```
def uses_only(word, available):
    return set(word) <= set(available)
```

操作符<= 检查一个集合是否是另一个集合的子集，包括两个集合相等的情况。这正好符合word 中所有字符都出现在available 中。

19.6　计数器

计数器（counter）和集合类似，不同之处在于，如果一个元素出现

超过一次，计数器会记录它出现了多少次。如果你熟悉多重集（multiset）这个数学概念，就会发现计数器是多重集的一个自然的表达方式。

计数器定义在标准模块collections 中，所以需要导入它再使用。可以用字符串、列表或者其他任何支持迭代访问的类型对象来初始化计数器：

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r':2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

计数器有很多地方和字典相似。它们将每个键映射到其出现次数。和字典一样，键必须是可散列的。

但和字典不同的是，在访问计数器中不存在的元素时，它并不会抛出异常。相反，它会返回0 ：

```
>>> count['d']
0
```

我们可以使用计数器来重写练习10-6中的is_anagram 函数：

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

如果两个单词互为回文，则它们会包含相同的字母，且各个字母的计数相同，所以它们对应的计数器对象也会相等。

计数器提供方法和操作符来进行类似集合的操作，包括集合加法、减法、并集和交集。计算器还提供一个非常常用的方法most_common，它返回一个值-频率对的列表，按照最常见到最少见来排序:

```
>>> count = Counter('parrot')
>>> for val, freq in count, most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7 defaultdict

collections 模块还提供了defaultdict，它和字典相似，不同的是，如果你访问一个不存在的键，它会自动创建一个新值。

创建一个defaultdict 对象时，需要提供一个用于创建新值的函数。用来创建对象的函数有时被称为工厂（factory）函数。用于创建列表、集合以及其他类型对象的内置函数，都可以用作工厂函数:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

请注意，参数是list（一个类对象），而不是list()（一个新的列表）。你提供的函数直到访问不存在的键时，才会被调用的：

```
>>> t = d['new key']
>>> t
[]
```

新列表t也会加到字典中。所以，如果我们修改t，改动也会在d中体现：

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

如果创建一个由列表组成的字典，使用defaultdict往往能够帮你写出更简洁的代码。在练习12-2的解答中，我创建了一个字典，将排序的字母字符串映射到可以由那些字母拼写出来的单词列表。例如，'opst'映射到列表['opts', 'post', 'pots', 'spot', 'stop', 'tops']。可以从http://thinkpython2.com/code/anagram_sets.py下载该解答。

下面是原始的代码：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

这个函数可以用setdefault 简化，你可能在练习11-2中也用过：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

但这个解决方案有一个缺点，它不管是否需要，每次都会新建一个列表。对于列表来说，这并不算大问题，但如果工厂函数非常复杂，就有可能成为问题了。

我们可以使用defaultdict 来避免这个问题，并进一步简化代码：

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
```

```
        t = signature(word)
        d[t]. append(word)
    return d
```

在练习18-3的解答中，函数has_straightflush 中使用了setdefault 。可以从http:// thinkpython2.com/code/PokerHandSoln.py下载它。但这个解决方案的缺点是，不管是否必需，每次循环迭代都会创建一个新的Hand 对象。作为练习，请使用defaultdict 重写该函数。

19.8　命名元组

很多简单的对象其实都可以看作是几个相关值的集合。例如，第15章中定义的Point 对象，包含两个数字，即x 和y 。定义一个这样的类时，通常会从init 方法和str 方法开始：

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self,):
        return '(%g, %g)' % (self.x, self.y)
```

这里用了很多代码来传达很少的信息。Python提供了一个更简洁的方式来表达同一个意思：

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

第一个参数是你想要创建的类名。第二个参数是Point对象应当包含的属性的列表，以字符串表示。namedtuple 的返回值是一个类对象：

```
>>> Point
<class '__main__.Point'>
```

这里Point类会自动提供__init__ 和__str__ 这样的方法，所以你不需要写它们。

要创建一个Point对象，可以把Point类当作函数来用：

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

init 方法使用你提供的名字把实参值赋给属性。str 方法会打印出Point对象及其属性的字符串表示。

可以使用名称来访问命名元组的元素：

```
>>> p.x, p.y
(1, 2)
```

也可以直接把它当作元组来处理：

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

命名元组提供了快速定义简单类的方法，但其缺点是简单的类并不会总保持简单。可能之后你需要给命名元组添加方法。如果那样，可以定义一个新类，继承当前的命名元组：

```
class Pointier(Point):
    # 在这里添加更多的方法
```

或者也可以直接切换成传统的类定义。

19.9 收集关键词参数

在12.4节中，我们见过如何编写函数将其参数收集成一个元组：

```
def printall(*args):
     print(args)
```

```
```

可以使用任意个数的按位实参（也就是说，不带名称的实参）来调用这个函数：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

但是*号操作符并不会收集关键词实参：

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

要收集关键词实参，可以使用**操作符：

```
def printall(*args, **kwargs):
   print(args, kwargs)
```

这里收集关键词形参可以任意命名，但kwargs 是一个常见的选择。收集的结果是一个将关键词映射到值的字典：

```
>>> printall(1, 2.0, third='3')
(1, 2.0){'third': '3'}
```

如果有一个关键词到值的字典，就可以使用分散操作符** 来调用函数：

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

没有用分散操作符的话，函数会把d 当作一个单独的按位实参，所以它会把d 赋值给x ，并因为没有提供y 的赋值而报错：

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

当处理参数很多的函数时，创建和传递字典来指定常用的选项是非常有用的。

19.10 术语表

条件表达式 （conditional expression）：一个根据条件返回一个或两个值的表达式。

列表理解 （list comprehension）：一个以方框包含一个for 循环，生成新列表的表达式。

生成器表达式（generator expression）：一个以括号包含一个for循环，返回一个生成器对象的表达式。

多重集 （multiset）：一个用来表达从一个集合的元素到它们出现次数的映射的数学概念。

工厂函数 （factory）：一个用来创建对象，并常常当作参数使用的函数。

19.11 练习

练习19-1

下面的函数可以递归地计算二项式系数：

```
def binomial_coeff(n, k):
    """计算(n, k)的二项式系数.

    n: 试验次数
    k: 成功次数

    返回：int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

使用内嵌条件表达式来重写该函数。

注意：这个函数效率不高，因为它会不停地重复计算相同的值。可以通过使用备忘（memoizing，参见11.6节）来提高它的效率。但你可能会发现，使用条件表达式之后，添加备忘会变得比较困难。

第20章　调试

调试程序时，应当区分不同类型的错误，以便更快地查找出错误原因。

- 语法错误（semantic error）在将源代码翻译为字节码的过程中由解释器发现。它们通常表示有程序结构错误。例如，在`def`语句的末尾漏掉冒号，会产生一个有些冗余的错误信息`SyntaxError: invalid syntax`。
- 运行时错误（runtime error）由解释器在程序运行的过程中发现错误后产生。大部分错误消息都包含了错误发生的位置以及正在执行的函数的信息。例如：一个无限递归最终会导致运行时错误`maximum recursion depth exceeded`（超过最大递归深度）。
- 语义错误（semantic error）是程序运行中没有产生错误信息，但做的事情却不正确的情况。例如：一个表达式求值的顺序和你预想的不同，因此产生了不正确的结果。

调试的第一步就是弄清楚你面对的到底是哪种类型的错误。虽然下面的几节是按照错误类型来组织的，但有些技巧其实可以适用于多种情形。

20.1　语法错误

语法错误，在弄清楚它们是什么之后，通常都很容易修正。不幸的

是，错误信息往往没什么帮助。最常见的错误信息是`SyntaxError: invalid syntax`和`SyntaxError: invalid token`，这两种都没多少信息量。

另一方面，信息也确实告诉你问题在程序中发生的位置。实际上，它告诉你的是 Python发现错误的位置，而并不一定总和错误发生的位置相同。有时候错误发生在错误信息指明的位置之前，往往是前一行。

如果你递增地构建程序，应当很清楚错误发生的位置。它常常在你最后添加的那行代码上。

如果你是从书本中复制代码，则最好先仔细比较自己的代码和书中的代码。检查每一个字母。同时请记得书本也可能是错的，所以如果你看到一个像是语法错误的东西，那么它有可能就是。

下面是一些可以避免最常见的语法错误的方法。

1．确保你没有使用Python关键字作为变量名称。

2．检查在每一个复合语句的语句头结尾，都有一个冒号，包括`for`、`while`、`if`和`def`语句。

3．确保程序中每个字符串都有前后匹配的引号。确定每个括号都是直引号（如"），而不是弯引号（如”）。

4．如果有三引号（单引号或双引号字符）多行字符串，确保你正确结束了字符串。没有正确结束的字符串，会导致程序结尾处产生`invalid token`错误，或者它会将接下来的程序看作字符串的一部

分，直到遇到下一个字符串为止。这种情况下，可能都不会产生错误信息！

5．没有关闭的开始符号（(、{ 或[）会让Python继续解析下一行，并当作当前语句的一部分。通常来说，会在下一行立即产生一个错误。

6．检查在条件判断时将‘==’写成‘=’的经典错误。

7．检查缩进，确保它们是按照设想正确排布的。Python可以处理空格和制表符，但如果混合使用它们，则可能产生问题。避免这种问题最好的办法是使用一个懂得Python的编辑器，并由它产生一致的缩进。

8．如果你的代码中有非ASCII字符（包括字符串和注释中），虽然Python 3通常能处理好非ASCII字符，但还是可能导致问题。当你从网页或其他来源直接复制文本时，需要格外注意。

如果上面的办法都没用，请继续看下一节。

我一直进行修改，但没有什么区别

如果解释器报出一个错误而你又找不到，有可能是因为解释器和你用的并不是同一套代码。检查你的编程环境，确保你正在编辑的代码和Python运行的是同一个。

如果不确定，可以尝试在程序开头加上一个明显而故意的错误。再运行一次。如果解释器并没有发现新的错误，那么说明你运行的不是新代码。

可能有以下几种原因。

- 你编辑了代码，但忘了保存更改就直接运行了。有的编程环境会帮你自动保存，有的不会。
- 你修改了文件名，但仍然在使用旧文件名运行程序。
- 你的编程环境可能没有正确配置。
- 如果你在编写一个模块，并使用`import`，请确保你的模块名称没有和Python标准模块冲突。
- 如果你在使用`import`来读入模块，请记得重载一个修改过的文件时，需要重启解释器或者使用`reload`。如果你直接重新导入这个模块，它并不会做任何事。

如果你遇到困难被卡住，而且弄不清楚到底怎么回事，一个办法是重新以最简单的类似“Hello，World！”的程序开始，并确保你能让一个已知的程序正确运行。然后逐渐添加原先程序的部分到新的程序中。

20.2　运行时错误

一旦你的程序已经确保语法正确，Python可以读它，并且至少可以开始运行它。这时候可能发生哪些错误？

20.2.1　我的程序什么都不做

这个问题最常见的原因是你的文件包含了各种函数和类的定义，但没有实际调用函数来启动执行。如果你是为了导入模块使用它们提供的类和函数，那么这么做可能是故意的。

如果不是故意的，则确保在程序中有一个函数调用，并确保执行流程能到达这一函数调用（参见20.2.5节）。

20.2.2　我的程序卡死了

如果一个程序突然停止并看起来什么事情都没做，它就“卡死了”。通常这意味着程序掉入一个死循环或者无限递归中。

- 如果怀疑一个特别的循环可能是问题所在，可以在循环开始前添加一个`print`语句，打出“进入循环”，在循环结尾处之后也添加一个，打出“退出循环”。

再次运行程序。如果你看到第一个输出，而没有看到第二个，说明你确实遇到一个死循环了。无限循环的内容参见20.2.3节。

- 大部分情况下，无限递归都会让程序运行一会儿，然后产生“RuntimeError: Maximum recursion depth exceeded”错误。如果发生这种情况，参见20.2.4节。

如果你没有看到这个错误，但怀疑可能是递归方法或函数产生的问题，也同样可以使用20.2.4节中的技巧。

- 如果上面两步都没用，尝试其他循环或其他递归方法与函数。
- 如果这些都没用，说明可能是你没理解你的程序的执行流程。执行流程的内容参见20.2.5节。

20.2.3　无限循环

如果你觉得有一个无限循环并知道是哪个循环导致的问题，可以在

循环的结尾处添加一个print 语句，打印出循环条件中的变量值，以及条件的值。

例如：

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

现在当你再次运行程序时，能够看到每次循环中打印出的3行输出。最后一次循环时，条件应该变为False 。如果循环一直进行，你应当可以看到x 和y 的值，并可能弄清楚为什么它们没有被正确更新。

20.2.4　无限递归

大部分情况下，无限递归会导致程序运行一会儿，然后产生Maximum recursion depth exceeded 的错误。

如果你怀疑一个函数导致了无限递归，保证递归确实有一个基准情形。应该有一个条件能导致函数直接返回而不再继续递归调用。如果没有，那么你可能需要重新思考算法，并定位一个基准情形。

如果有一个基准情形，但程序似乎没有到达它，可以在函数的开头加一个print 语句来打印参数。现在当你重新运行程序时，会看到每次函数调用时都会打出几行输出，并能看到每次调用的参数值。如果参数

并没有向基准情形变化，你大概能发现为何如此。

20.2.5　执行流程

如果你不确认程序中的执行流程如何走向，可以在每个函数的开头添加一个`print`语句，打印类似“进入函数`foo`”之类的输出。这里`foo`是函数名。

现在如果你重新运行程序，它会打印出每个函数调用的轨迹。

20.2.6　当我运行程序，会得到一个异常

如果在运行时遇到一个问题，Python会打印出一个信息，包含错误的名称，程序中发生这个错误的位置，以及一个回溯。

回溯里标明了当前执行的函数，以及调用它的函数，以及调用这个调用者的函数，依此类推。换句话说，它回溯了从程序开头直到错误发生所在位置的整个调用轨迹，包括了每个函数所在文件中的行号。

第一步是检查程序中错误发生的位置，并尝试弄清楚问题所在。下面是一些常见的运行时错误。

NameError

你在试图使用一个当前环境中并不存在的变量。检查变量名是否有拼写正确，或至少是一致。请记得局部变量是局部的，不能在定义它们的函数之外使用。

TypeError

有3种可能的原因。

- 你在尝试错误地使用一个值。例如，使用不是整数的值来索引字符串、列表或元组。
- 格式字符串中，内部的格式项和传入的参数不匹配。当格式项的数目不对或者转换的类型不对时都可能发生。
- 调用函数时使用了错误数量的参数。对于方法来说，查看方法定义并检查第一个参数是否为`self` 。接着查看方法调用；确保你是在正确类型的对象上调用方法，并正确提供了其他参数。

KeyError

你在试图用一个字典并不包含的键来查找字典的元素。如果键是字符串，请注意大小写问题。

AttributeError

你在尝试访问一个并不存在的属性或方法。检查拼写！你可以使用内置的`vars` 函数来列出存在的属性。

如果AttributeError指明一个对象是`NoneType` ，则意味着它是`None` 。那么问题不是属性名而是对象。

对象为None的原因可能是你忘了从函数里返回值；如果函数执行到结尾都没有遇到`return` 语句，那么它会返回`None` 。另一个常见的原因是使用了一个返回`None` 的列表方法作为结果，如`sort` 。

IndexError

你在访问列表、字符串或元组时使用的索引大于它的长度减一。在错误发生的前一行，添加一个`print` 语句展示索引的值和数组的长度。数组长度是否正确？索引大小是否正确？

Python调试器（`pdb` ）在查找异常时很有用，因为它让你可以在错误发生之前的地方查看程序的状态。可以在http://docs.python.org/3/library/pdb.html阅读`pdb` 的相关资料。

20.2.7　我添加了太多print 语句，被输出淹没了

使用`print` 语句进行调试的问题之一是你可能被太多的输出所埋没。有两种方法可以继续：简化输出，或者简化程序。

要简化输出，可以删除或注释掉没用的`print` 语句，或者将它们合并起来，或者格式化输出让它们更容易看懂。

要简化程序，有几件事情可做。首先，简化程序所处理的问题。例如，如果你在搜索一个列表，就改为搜索一个很小的 列表。如果程序从用户获得输入，则输入可以产生错误的最简单的输入。

其次，清理程序。删除无效代码，并重新组织代码让它尽可能更可读。例如，如果你怀疑问题出在程序的一个很深的嵌套部分中，则应当尝试重写那部分，让它的结构更简单。如果你怀疑一个很大的函数，则尝试将它拆分为多个更小的函数，并分别测试它们。

找寻最简测试用例的过程往往能带你找到问题所在。如果发现程序在一种情况下正常工作，而在另一种情况下则不能，那这些情况本身就给你一些线索。

类似地，重写一部分代码可以帮你找到细微的bug。如果你做出一个认为不该影响程序的改变，而它确实出问题了，这就给了具体的提示。

20.3　语义错误

从某种角度看，语义错误更难调试，因为解释器并不提供任何信息。只有你自己知道程序到底应该怎么做。

解决语义错误的第一步是在程序文本和你看到的程序行为之间建立一个连接。你需要对程序实际在做什么有一个假设。让这件事情很难的原因之一是计算机运行得太快。

你常常会希望程序能够减慢到人的速度，而使用调试器时你可以做到。但往程序里插入几条精确放置的`print`语句，比起设置调试器，插入或删除断点，并“单步”执行到程序出错的地方，往往花费的时间更少。

20.3.1　我的程序运行不正确

你应该问自己如下几个问题。

- 程序中有没有地方你期望它去做而实际上没有发生的？找到运行那段功能的代码，并确保它确实如你所期望的那样运行了。
- 有没有一些不应该发生的事情？找到程序中运行了某种不该出现的功能的代码。
- 有没有一段代码产生的效果和你所期望的不一致？确保你完全明白

该段代码，特别是当它牵涉到其他Python模块的函数或方法时。阅读你调用的函数的文档。使用简单的测试用例测试它们并检查结果。

为了能够编程，你需要程序如何工作的一个思维模型。如果编写出一段和你预期不同的代码，常常问题不是在程序本身，而是在你的思维模型上。

修正你的思维模型的最佳方法是将程序划分成不同部分（通常是函数和方法）并独立测试每一个部分。一旦找到你的模型和真实世界的偏差，就能够解决问题了。

当然，在开发程序时你应当分组件进行构建和测试。如果发现一个问题，应该只需要检查一小部新的不确认是否正确的代码。

20.3.2　我有一个巨大而复杂的表达式，而它和我预料的不同

编写复杂的表达式并没有问题，只要能保证它们还可读。但它们也会变得更难调试。将复杂的表达式拆分成一系列的赋值到临时变量的语句，常常是个好主意。

例如：

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

这个表达式可以写作：

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

后面更清晰的版本也更加可读，因为变量名称提供了附加的文档信息，它也更加容易调试，因为你可以检查中间变量的类型，并打印它们的值。

复杂表达式的另一个问题是求值的顺序可能和你所期望的不同。例如，如果你将表达式$x /2\pi$翻译成Python，可能会这么写：

```
y = x / 2 * math.pi
```

这样并不正确，因为乘法和除法有相同的优先级，并且语句求值的顺序是从左至右。所以这个表达式计算的实际上是$x \pi/2$。

调试表达式的一个好办法是添加括号来显式控制求值顺序：

```
y = x / (2 * math.pi)
```

任何时候如果不确定求值的顺序，都可以使用括号。这样不但会让程序更加正确（从按照你的设想来做的角度说），也会让其他人更容易阅读你的代码，因为不需要去记忆操作的顺序。

20.3.3　我有一个函数，返回值和预期不同

如果你在程序中有return 语句返回一个复杂的表达式，则没有机会在返回之前打印结果。这时候，也可以使用临时变量。例如，这个语句：

```
return self.hands[i].removeMatches()
```

可以写作：

```
count = self.hands[i].removeMatches()
return count
```

现在你有机会在返回之前显示count 的值了。

20.3.4　我真的真的卡住了，我需要帮助

首先，试着离开计算机几分钟。计算机会发射辐射影响大脑，产生下列症状。

- 挫败感和愤怒感。
- 迷信的信念（“我的计算机恨我”）和神奇的想法（“程序只有在我反戴帽子时才正确运行”）。
- 随机行走编程（尝试着写下所有可能的程序，并选择运行正确的那个）。

如果你发现自己正在遭受这些症状之一，请马上站起来出去散个步。当你平静下来后，再思考程序。它在做什么？产生那种行为的可能原因有哪些？上一次程序还正确运行是什么时候，之后你做了什么？

有时候发现一个bug确实需要时间。我常常能够在远离计算机并让思维休息之后找到bug。找到错误的最佳地点有火车上、浴缸中及将要入睡之前在床上。

20.3.5 不行，我真的需要帮助

这种事确实会发生。即使最好的程序员也会偶尔卡住。有时候你在一段程序上工作太久了所以反而看不到错误。你需要一双新的眼睛。

在叫人帮忙之前，请确保你已经准备好。你的程序应当尽量简单，而你应当使用最小的输入来复现错误。你应当在合适的地方放好了`print`语句（并且它们的输出应当容易理解）。你应当足够理解这个问题，因此能够简明扼要地描述它。

当你找人帮忙时，请确保给他们需要的信息。

- 如果有错误信息，它是什么，它代表了程序的哪部分？
- 在这个错误发生之前，你做的最后一件事情是什么？你写的最后一段代码是什么？失败的新测试用例是什么？
- 目前为止你做了哪些尝试，并从中得到了什么？

当你找寻bug时，思考一下如何做才能找得更快。下一次见到类似的情形时，就能够更快地找到问题了。

记住，目标不只是让程序正确运行。目标是学会如何让程序正确运行。

第21章　算法分析

这个附录编选自O’Reilly Media出版的Allen B. Downey的*Think Complexity*（2012）一书。当你读完本书之后，可能会想继续读那本书。

算法分析 是计算机科学的一个分支，研究算法的性能，尤其是它们的运行时间和空间需求。参见http://en.wikipedia.org/wiki/Analysis_of_algorithms。

算法分析的实践目标是预测不同算法的性能，以便于指导设计决策。

在2008年的美国总统大选中，候选人巴拉克·奥巴马在访问Google公司时被要求做一个即兴分析。Google的首席执行官埃里克·施密特问他“给100万个32位整数排序的最高效算法”是什么。奥巴马显然被提示了，因为他马上回答，“我觉得冒泡排序可能是错误的做法”。参见http://bit.ly/1MpIwTf。

这是真的：冒泡排序在概念上很简单，但对于大数据量的排序很慢。施密特想得到的答案可能是“基数排序”（http://en.wikipedia.org/wiki/Radix_sort）[1]。

算法分析的目标是在不同算法间做出有意义的比较，但也有一些问题。

- 算法的相对性能可能依赖于硬件的特征，所以一个算法可能在机器A上更快，另一个在机器B上更快。这个问题的通用解决方法是先指定一个机器模型，并分析在一个指定的机器模型中一个算法需要执行的步骤或操作。
- 相对性能还可能依赖于数据集的细节特征。例如，有的排序算法在数据已经是部分排序的情形下比其他算法更快，有的程序在这种情况下反而慢。避免这个问题的通常办法是分析最坏情况 场景。有时候分析平均情况的性能也有用，但也通常会更难，因为有哪些情形可以用来“平均”往往并不明显。
- 相对性能也依赖于问题的规模。对小序列更快的排序算法可能对大序列就慢了。这个问题的通常解决方案是用一个问题规模的函数来表达运行时间（或操作数），并根据问题规模增大的速度将函数进行归类。

这种比较的好处之一是自然而然地可以将算法进行简单地分类。例如，如果我知道算法A的运行时间趋向于和输入的规模n 成比例，而算法B趋向于和n^2 成比例，那么我会预期至少对于大的n 值，算法A比算法B快。

这种分析也有需要注意的地方，后面会谈到。

21.1 增长量级

假设你需要分析两个算法，并依照输入的规模来表达它们的运行时间：算法A需要$100n+1$步来解决规模为n 的问题，算法B需要n^2+n+1步。

下面的表格显示了这两个算法在不同的问题规模下的运行时间：

输入规模	算法**A**的运行时间	算法**B**的运行时间
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$>10^{10}$

在n =10时，算法A看起来很差；它几乎需要10倍于算法B的时间。但对于n =100来说它们就已经差不多了，而在更大的规模时，算法A远好于算法B。

这里根本的原因在于对很大的n 值，任何包含n^2 项的函数都会比首项是n 的函数增长快速很多。首项 是一个多项式中最高次方的项。

对于算法A，首项有一个很大的系数100，因此算法B在小的n 时比算法A快。但不论系数是多少，总有一个n 值会导致$an^2 > bn$ 。

对于非首项来说也如此。即使算法A的运行时间是n +1000000，对于足够大的n ，仍然会比算法B快。

总的来说，我们预期有更小的首项的算法对大规模问题来说是更好的算法。但对于小一些的问题来说，可能存在一个交叉点 ，那里其他

算法可能更好。交叉点的位置取决于算法的细节、输入以及硬件的条件，所以在算法分析时常常被忽略掉。但那并不意味着你可以忘记它。

如果两个算法有相同的首项，则很难说哪一个更好；同样地，答案也取决于细节条件。所以对于算法分析来说，首项相同的函数被认为是同等的，即使它们的系数不同。

增长量级 就是各种增长行为被认为是同等的函数的集合。例如，$2n$ 、$100n$ 和$n+1$都是一个增长量级，用大**O**标记法 写作$O(n)$，通常称为线性的 ，因为这个集合中的每个函数都依据n 线性增长。

所有首项是n^2 的函数都属于$O(n^2)$，它们被称为是平方的 。

下面的表格显示了算法分析中大部分最常见的增长量级，按照更坏的程度递增：

增长量级	名称
$O(1)$	常量级
$O(\log_b n)$	对数级（对任意b ）
$O(n)$	线性级
$O(n \log_b n)$	$n \log n$
$O(n^2)$	平方级

$O(n^3)$	立方级
$O(c^n)$	指数级（底数c任意）

对于对数项，底数并没有影响；修改底数相当于乘以一个常量，而那样并不影响增长量级。类似地，所有的指数函数都是同一个增长量级，不论指数的底数是什么。指数函数增长非常迅速，所以指数级算法只在小规模问题中应用。

练习**21-1**

在http://en.wikipedia.org/wiki/Big_O_notation上阅读大O标记法的维基百科页面，并回答下列问题。

1．$n^3 + n^2$的增长量级是多少？$1000000n^3 + n^2$呢？$n^3 + 1000000n$呢？

2．$(n^2 + n)\cdot(n + 1)$的增长量级是多少？在相乘之前，请记住你只需要首项。

3．如果f是$O(g)$，对于未指定的函数g，我们怎么说$af+b$？

4．如果f_1和f_2都是$O(g)$，那么f_1+f_2呢？

5．如果f_1是$O(g)$而f_2是$O(h)$，那么f_1+f_2呢？

6．如果f_1是$O(g)$而f_2是$O(h)$，那么$f_1\cdot f_2$呢？

关心程序性能的程序员常常会觉得这种分析很难理解。他们有道

理：有时候系数和非首项也能带来不同。有时候硬件的细节、编程语言，以及输入的特征，都能带来很大的区别。并且对于小规模问题来说，渐进行为是无关要紧的。

但如果在脑中记着这些需要注意的要点的话，算法分析毕竟是一个有用的工具。至少对于大规模问题来说，“更好”的算法往往确实更好，并且有时候它会好得多 。两个增长量级相同的算法的区别往往是一个常量值，但一个好算法和一个坏算法的差距是没有界限的！

21.2　Python基本操作的分析

在Python中，大部分算术操作都是常量时间的；乘法通常比加法和减法花费更多时间，而除法花费的更多，但这些操作的时间与参数的大小无关。特别大的整数是一个例外，在那种情况下，运行时间随着数字的位数增加而增加。

索引操作——在序列或字典中读写元素——也是常量时间的，与数据结构的规模无关。

遍历一个序列或字典的for 循环通常是线性的，只要循环体内的操作本身是常量级。例如，将一个列表的元素相加是线性的：

```
total = 0
for x in t:
    total += x
```

内置函数sum 也是线性的，因为它做相同的事情。但它趋向于更快些，因为实现得更高效；用算法分析的语言来说，就是它有一个更小的首项系数。

作为一个经验规则，如果循环体的增长量级是$O(n^a)$则整个循环是$O(n^{a+1})$。例外情况是当你能够证明循环在一个常量数的迭代之后就能退出。如果不论n 是多少，循环只最多运行k 次，则即使对很大的k 来说，整个循环的增长量级还是$O(n^a)$。

乘以k 并不会改变增长量级，而除法也不会。所以，如果一个循环体的增长量级是$O(n^a)$，那么它运行n/k 次，即使对很大的k 来说，整个循环的增长量级也仍然是$O(n^{a+1})$。

大部分字符串和元组操作都是线性的，只有下标访问和len 函数例外，它们是常量级时间的。内置函数min 和max 是线性的。切片操作的运行时间与输出的长度成正比，而与输入的长度无关。

字符串拼接是线性的，它的运行时间与操作数的长度的总和有关。

所有的字符串方法都是线性的，但如果字符串的长度受限于一个常量（例如，在只有一个字符的字符串的操作），可以看作是常量的。字符串方法join是 线性的，它的运行时间与字符串的总长度有关。

大多数列表方法是线性的，但也有一些例外。

- 在列表结尾处添加一个元素的操作平均来说是常量时间的；当它空间不足时，偶尔会复制到另一个更大的地方，但总的n 次操作的时间量级是$O(n)$，所以每次操作的平均时间是$O(1)$。

- 从列表结尾删除一个元素的操作是常量时间的。
- 排序的量级是$O(n \log n)$。

大部分字典操作和方法都是常量时间的，但也有一些例外。

- `update` 的运行时间和作为参数传入的字典的大小成比例，而不是被更新的字典本身。
- `keys` 、`values` 和`items` 都是常量时间，因为它们返回的是迭代器。但是，如果循环遍历这个迭代器，则循环是线性的。

字典的效率是计算机科学的一个小奇迹。我们会在21.4节中介绍它是如何工作的。

练习**21-2**

在http://en.wikipedia.org/wiki/Sorting_algorithm阅读排序算法的维基百科页面并回答下列问题。

1．什么是“比较排序”？比较排序的最坏情况的增长量级最好是什么？任何排序算法中，最坏情况的增长量级最好是多少？

2．冒泡排序的增长量级是多少？为什么奥巴马认为它是“错误的做法”？

3．基数排序的增长量级是多少？要使用它，我们需要哪些前置条件？

4．稳定排序是什么，为什么在实践中它很重要？

5．最差的（有名字的）排序算法是什么？

6．C语言库里用的排序算法是什么？Python里用的是什么？这些算法稳定吗？你可能需要去Google搜索这些答案。

7．很多非比较排序都是线性的，那么为什么Python会使用$O(n \log n)$的比较排序呢？

21.3 搜索算法的分析

搜索 是一种算法，接收一个集合和一个目标元素，并决定这个元素是否在集合中，通常返回元素的索引。

最简单的搜索算法是“线性搜索”，即按顺序遍历集合的每一个元素，直到找到目标元素为止。在最坏的情况下，它会遍历整个集合，所以运行时间是线性的。

序列的`in` 操作符使用一个线性搜索；字符串方法`find` 和`count` 也是这样。

如果序列中的元素是排好序的，可以使用二分查找，它的增长量级是$O(\log n)$。二分查找和在字典（真实的字典，而不是那个数据结构）中查找单词的算法类似。不像普通搜索那样从第一个元素开始，它是从序列的中间开始，检查要查找的词是在中间的元素之前还是之后。如果在之前，则继续查找序列的前半段，否则查找后半段。不论哪种情况，都可以将查找的数量减少一半。

如果序列有1 000 000个元素，大概需要花20个步骤找到单词或者发现它不存在。所以那样会比线性查找快大概50 000倍。

二分查找可以比线性查找快很多，但需要序列本身是排好序的，也就需要一些额外工作。

有另一个数据结构，称为散列表（hashtable），它甚至更快——它可以用常量时间来搜索——而且不需要元素是排好序的。Python字典是使用散列表实现的，因此大部分字典操作，包括`in`操作符，都是常量时间的。

21.4　散列表

为了解释散列表的工作机制以及为何它的效率如此好，我们先从一个简单的映射实现开始，并逐步改善它，直到成为一个散列表。

我使用Python来展示这些实现。但真实世界中，你不需要用Python写这样的代码，你只需要直接使用字典即可！所以本章中剩下的部分，你需要想象字典并不存在，而你需要实现一个数据结构将键映射到值。你需要实现的操作有以下几个。

`add(k，v)`

添加一个新项，将键`k`映射到值`v`。在Python字典`d`中，这个操作写作`d[k] = v`。

`get(k)`

根据键k 查找对应的值。在Python字典d 中，这个操作写作d[k] 或d.get(k) 。

就现在来说，我假设每个键只出现一次。最简单的实现是使用一个元组列表，每个元组是一个键值对：

```
class LinearMap:

    def _ _init_ _(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

add 往元组列表中添加一项，这个操作是常量时间的。

get 使用一个for 循环来搜索列表：如果找到了目标键，则返回对应的值；否则抛出KeyError 。所以get 是线性的。

另一个方案是让列表按照键来排序。这样get 就可以使用二分查找，其增长量级是$O(\log n)$。但插入一个新项到列表中间是线性的，所以这可能也不是最好的选择。也有数据结构可以用对数时间实现add 和get ，但那仍然没有常量时间好，所以我们继续。

改善LinearMap 的方法之一是将键值对的列表拆分成更小的列表。

下面是一个称为BetterMap 的实现，它是一个包含100个LinearMap的列表。我们接下来会看到，get 的增长量级仍然是线性的，但是BetterMap 离散列表更近了一步。

```
class BetterMap:

    def _ _init_ _(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

__init __创建由n 个LinearMap 组成的列表。

find_map 被add 和get 调用，用来确定用哪个映射来保存新项，或者到哪个映射里去搜索。

find_map 使用了内置函数hash ，它接收几乎所有的Python对象，并返回一个整数。这个实现的限制之一是它只对可散列的键类型可用。可变类型，如列表和字典，是不可散列的。

两个认为相等的可散列对象会返回相同的散列值，但反过来并不一

定是真：两个具有不同值的对象可以返回相同的散列值。

find_map 使用求余操作符来将散列值封装到0到len(self.maps)的范围中，这样结果是列表的一个合法索引。当然，这意味着很多不同的散列值会封装到同一个索引上。但如散列函数将对象分配地很均匀（这也是散列函数设计的目标），那么我们预计每个LinearMap有*n* /100个项。

因为LinearMap.get 的运行时间是和其包含的项数成比例的，所以我们预计BetterMap会比LinearMap快100倍。增长量级仍然是线性，但首项系数更小。这很好，但仍然不如散列表好。

下面（终于）是让散列表能变快的关键原因：如果你能保证LinearMap的长度有限，LinearMap.get 则会是常量时间。你需要做的只是记录元素的总数，并当每个LinearMap的大小超过一个阈值时，重新划分散列表，添加更多的LinearMap。

下面是一个散列表的实现：

```
class HashMap:

    def _ _init_ _(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1
```

```
    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

每个HashMap 都包含一个BetterMap； __init __从2个LinearMap开始，并初始化num ，它会用来记录总的项数。

get 只需要分配到对应的BetterMap 。真正的工作都发生在add 中，它会检查项数和BetterMap 的大小：如果相等，那么每个LinearMap的平均项数是1，所以它调用resize 。

resize 创建一个新的BetterMap ，比之前大一倍，并将旧有的映射中的项“重新散列”到新的映射中。

重新散列是有必要的，因为LinearMap的数量的改变，导致find_map 的求余操作符的分母改变。也就是说，有些原先会散列到同一个LinearMap的项会分配到不同的LinearMap中（这也是我们想要的，对吧？）。

重新散列是线性的，所以resize 是线性的，看起来可能不好，因为我保证过add 应当是常量时间的。但请记得我们并不是每次都需要进行resize ，所以add 通常是常量时间的，只是偶尔会线性。add 运行*n* 次的总时间是和*n* 成比例的，因此每次调用add 的平均时间是常量时

间！

要明白散列表如何工作，考虑从一个空的HashTable开始，并添加一些项。我们从2个LinearMap开始，所以最开始两个add 会很快（不需要resize ）。我们说它们每次花费一单位的工作量。下一个add 会需要resize ，所以我们需要重新散列前两项（我们说这需要再加2个单位的工作量）并添加一个新项（再加1个单位）。再添加一项花费1单位，所以至今为止是4项花费了6个单位的工作。

下一个add 需要5个单位，但接着的3个都只需要1个单位，所以总共是8项花费了14单位。

再下一个add 需要9个单位，但接着我们可以在再次resize 之前添加7项，所以总共是16个add 花费了30单位。

在32个add 时，总共的花费是62单位，而我希望你已经开始看到其中的模式了。在n 个add 之后，假设n 是2的乘方，总的花费是$2n-2$单位，所以平均每个add 的工作量是稍微小于2个单位的。当n 是2的乘方时，这是最好情况；对于其他的n 值，平均工作量稍高一点，但这并不重要。重要的是这是$O(1)$。

图21-1用图形化的方式展示了这个过程。每个方块代表一个单位的工作量。每一列显示每个add 的工作量：从左到右，前两个add 花费1单位，第三个花费3单位，等等。

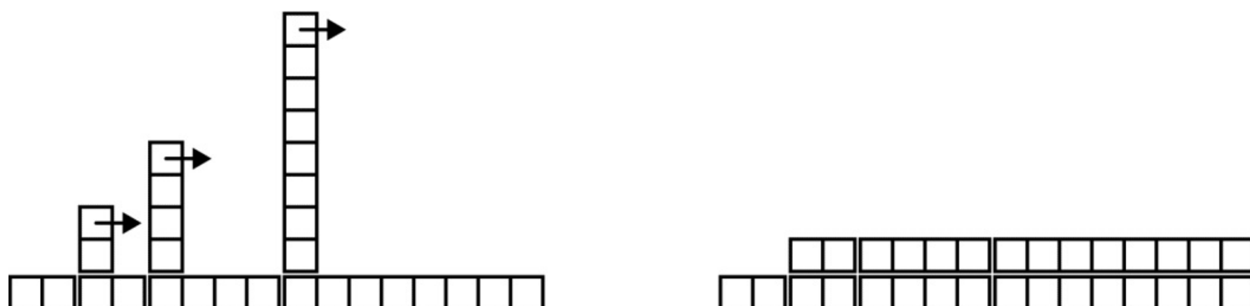


图21-1　散列表add的消耗

多余的重新散列的工作看起来像一序列不断增高的塔，之间的间隔越来越远。现在如果你将塔推倒，将`resize`的花费均摊到所有`add`操作上，就会发现n个`add`之后总的花费是$2n-2$。

这个算法的一个重要特点是当我们调整HashTable的大小时，它会几何增长；也就是，我们乘以一个常量到大小上。如果算术地增加大小——每次添加固定数量的数——那么每个`add`的平均时间是线性的。

可以从http://thinkpython2.com/code/Map.py下载我的HashMap实现，但请记住并没有使用它的理由。如果需要一个映射，直接用Python字典即可。

21.5　术语表

算法分析（analysis of algorithms）：通过对比运行时间以及/或者空间需求来对比算法的方法。

机器模型（machine model）：用于描述算法的简化的计算机表示形式。

最坏情况 （worst case）：让指定算法运行最慢（或者需要最多空间的）的输入。

首项 （leading term）：在多项式中，指数最高的项。

交叉点 （crossover point）：两个算法需要相同的运行时间或空间的问题规模。

增长量级 （order of growth）：在算法分析时，如果我们认为一组函数的增长速度可以看作相等，则将这组函数称为同一个增长量级的。例如，所有线性增长的函数都属于同一个增长量级。

大O表示法 （Big-Oh notation）：表达增长量级的方法。例如，$O(n)$表示所有线性增长的函数集合。

线性 （linear）：运行时间和问题规模（至少对于大规模来说）成正比的算法。

平方量级 （quadratic）：运行时间和n^2成正比的算法，其中n指的是问题规模。

搜索 （search）：定位集合（如列表或字典）中某个元素或者判定它不在其中的问题。

散列表 （hashtable）：一种表示键值对集合且搜索是常量级的数据结构。

---

[1] 但如果你在面试时被问到这个问题，我觉得更好的答案是：“给100

万个数排序的最快算法应当是使用我用的语言提供的排序函数。它的性能应当对绝大多数应用都足够好了，但如果发现我的程序太慢，我会使用一个性能分析器去查看时间花在哪里。如果看起来更快的排序算法会带来明显的提升，那我会去寻找一个基数排序的良好实现。”

译后记

《像计算机科学家一样思考》这一系列书，早有耳闻，它可谓开创了程序设计入门书的一个新思路。授人以鱼，不若授人以渔；教人编程，不如引导人思考；教人语言细节，不若指明语言精要。而结合Python语言之后，得到的《像计算机科学家一样思考Python》这本书，则是在这个思路上走到了一个极致的佳作。

我是工作之后才开始接触Python的。在那之前一直使用C/C++、Java、C#等传统风格的语言，再看到Python，不免有耳目一新之感。为何以往觉得晦涩难懂的程序设计理念，在Python中却表达得这么简洁易懂？为何以往需要绞尽脑汁才能拼出来的大段代码，在Python里却只需要几个简单调用即可？为何繁复的集合操作，在Python中却只需要一行列表理解循环语句就完成了？为何Python的文档那么容易找，还可以使用交互模式轻松尝试？每次使用Python编写程序之后，总会感慨，当初初学程序设计语言的时候，如果教的是Python该多好。相信所有学过C/C++之后再接触Python这类语言的人，都会有相同的感受吧。

那么是什么原因让C/C++几乎垄断了程序设计语言的教材呢？我觉得更多的是历史惯性。在计算机科学教育开始普及的20世纪70、80年代，C语言正在其鼎盛时期，几乎所有的人都在用C开发程序，操作系统、软件、游戏几乎都是用C甚至汇编开发的。硬件性能的限制，让那些更抽象、更高阶的语言，无法普及开来。因此教学自然也使用它。久而久之形成了惯性，到了新世纪，程序设计的教学已经赶不上语言发展

的潮流了。我们的程序越来越复杂，越来越像人脑，而教学的语言仍然在使用高级语言中最贴近机器的C。而C++、Java、C#，虽然相对于C更抽象高阶，但由于这些语言设计的初衷仍是以扩展C为主，所以不过是在这一惯性上多走了五十步而已。

本书正是扭转这种矛盾局面的一个有益的尝试。《像计算机科学家一样思考》是对程序设计教学模式的真谛的领悟，而使用Python这种简洁强大的高阶语言，也正是这种新思路最贴切的贯彻。授人以渔，自然应当用最好的渔具；引导人思考，当然也应使用更贴近人的思路而不是机器思路的语言。Python在高阶语言中，是一个从理念和实际综合考量后非常合适的候选。

在翻译过程中我发现，本书不但思路很贴切其教学主旨，从行文和用例来看也非常浅显易懂。全书讲了非常多的程序设计理念，在读过之后却会觉得那些理念都很自然，大概也是因为作者苦心安排，前后穿插，让读者能循序渐进地明白每个程序设计理念是因为什么而出现的原因吧。这种风格，再配合上精心编辑的示例，用于介绍任何程序设计语言，都是非常合适的。

如果将来我的孩子愿意学习程序设计，我愿意用这本书教他。

这一版，将语言升级到Python 3，从而更加贴近语言发展的趋势。作者对章节内容和示例练习也做出了重新组织和调整，使得阐述行文更加通畅。

尽管我已尽最大努力争取译文准确、完善，但仍然难免有疏漏之处，如发现问题，欢迎批评指正。电子邮箱zhaopuming@gmail.com。

译者介绍

赵普明　毕业于清华大学计算机系，从事软件开发行业近10年。从2.3版本开始接触Python，工作中使用Python编写脚本程序，用于快速原型构建以及日志计算等日常作业；业余时，作为一个编程语言爱好者，对D、Kotlin、Lua、Clojure、Scala、Julia、Go等语言均有了解，但至今仍为Python独特的风格、简洁的设计而惊叹。

作者介绍

Allen Downey是欧林工程学院（Olin College of Engineering）的计算机科学教授。他曾在韦尔斯利学院（Wellesley College）、科尔比学院（Colby College）和加州大学伯克利分校（U.C. Berkeley）任教。他从加州大学伯克利分校获得计算机科学博士学位，并从MIT获得硕士和学士学位。

封面介绍

本书封面的动物是卡罗来纳鹦鹉，也叫卡罗来纳长尾鹦鹉（学名Conuropsis carolinensis）。这种鹦鹉分布于美国东南部，并且是一种栖息在墨西哥以北的大陆鹦鹉，它们最北曾一度到达纽约和大湖区，但主要分布在佛罗里达州到卡罗来纳州一带。

卡罗来纳鹦鹉主色是绿色，头部黄色，成熟时前额和两颊会出现一些橙红色的条纹。它的平均尺寸是31～33 cm。它叫声狂暴而巨大，并且在捕食过程中会喋喋不休。它居住在沼泽与河畔的树洞中。卡罗来纳鹦鹉是喜欢群居的生物，平时以小群体形式生活，在捕食时可以达到几百只。

不幸的是，这些捕食过程往往在农田的庄稼地里进行，农夫会射击它们，以免破坏庄稼。它们的群体特性让它们会集体救助受伤的鹦鹉，结果让农夫可以一次杀光整群鹦鹉。不但如此，它们的羽毛被用做妇女的帽饰，也有一些鹦鹉被作为宠物。这些因素组合起来，导致在19世纪晚期，卡罗来纳鹦鹉变得非常稀少，并且禽类疾病也加剧了它们的减少。到20世纪20年代，这个物种灭绝了。

今天，全世界的博物馆中保存了700多只卡罗来纳鹦鹉的标本。

很多O’Reilly的书封面上的动物都是濒危物种，它们全都对世界有重要意义。请访问animals.oreilly.com来了解如何帮助它们的信息。

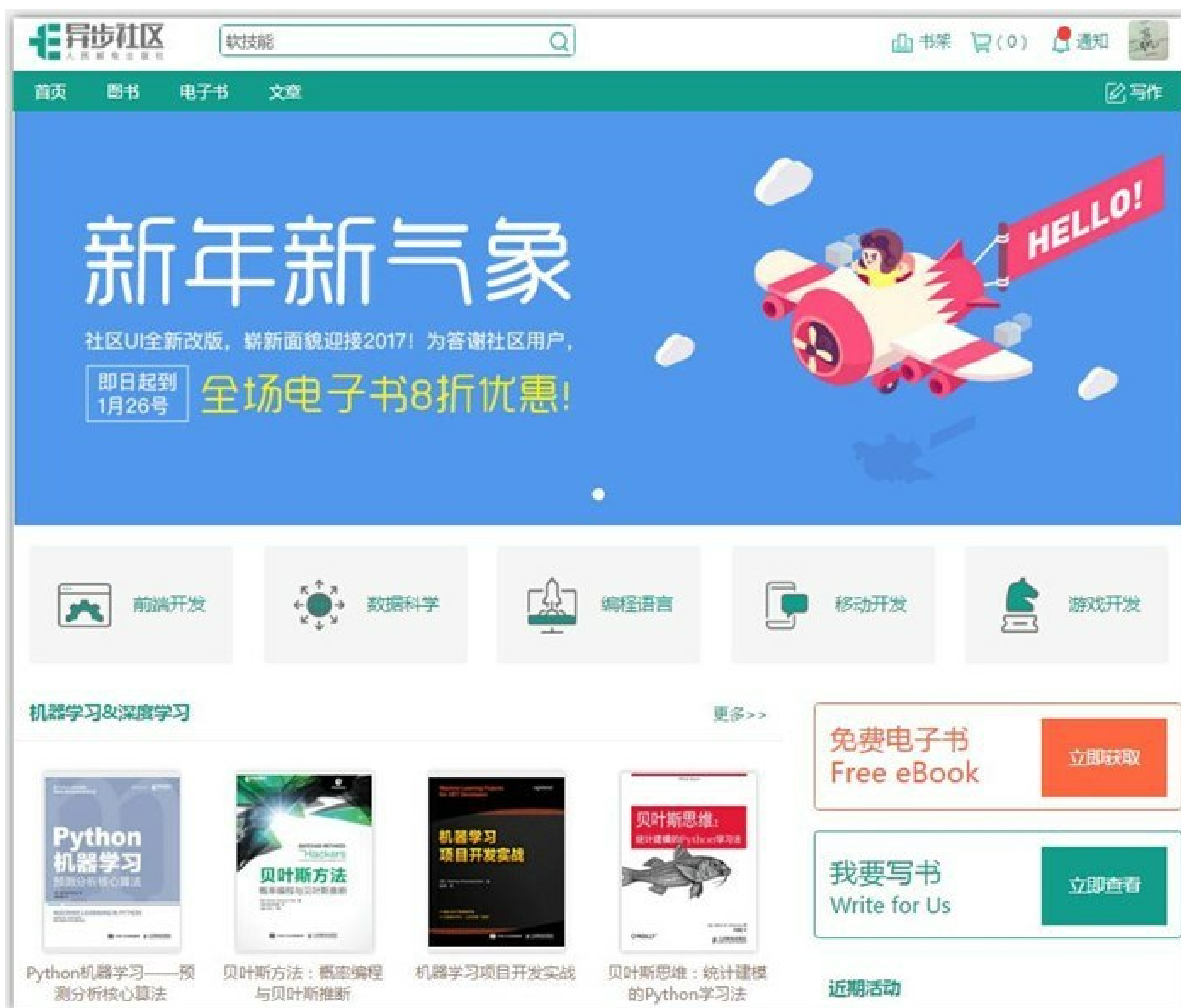
封面图片来自《约翰逊的自然历史》（*Johnson’s Natural History*）。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在 里填入可使用的积分数值，即可扣减相应金额。

特别优惠

购买本电子书的读者专享异步社区优惠券。 使用方法：注册成为社区用户，在下单购书时输入“**57AWG** ”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试

身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群：436746675

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿**&**咨询： contact@epubit.com.cn